

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.
wuuyang@cis.nctu.edu.tw

December 1, 2004

ABSTRACT

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 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 in an attempt to achieve better performance.

KEY WORDS

Java, assembly, translation, JIT, compilation ¹

1 Introduction

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, *JNI* [23, 15]).

An alternative is to create and manipulate objects purely in assembly code. This alternative approach may result in better performance. However,

¹The work reported here is supported by National Science Council, Taiwan, R.O.C., under grant NSC 93-2213-E-009-092.

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 [22] is an 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.

So far, there are two types of JIT implementation. One is fast, effective code generation [1]. It has no explicit intermediate representation, and it generates native code directly from bytecodes in a pass. Another one is optimizing compiler [4]. The optimizing compiler takes a conventional compilation approach that builds an intermediate representation (IR) and performs global optimizations based on the IR. There is also a JIT implementation that makes use of annotation of register allocation information [2].

Hsieh et al. [8, 9], who report on a byte code to native code translator, model the flow of translation and generate an intermediate representation. A few of native compilers, such as JBCC [10], do not incorporate intermediate representations. They use mapping styles to generate native code. Our translation system makes use of an enhanced mapping style to generate native code.

There are two types of Java bytecode optimization. One is to optimize the bytecode directly [14]. Apply well known optimizations to Java bytecodes. Another, such as SOOT [20], translates bytecode to intermediate representation, optimizes IR, and then generates new bytecode.

Some researchers [19, 17] propose native compilers for Java. Bothner [3] discussed a Java compiler based on GCC.

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. It builds the following six tables: the constant pool table, the `allclass-methodtable`, the branch table, the `jsr-ret` table, the exception table, and the instruction match table. The tables are used 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. Section 3 details the translation of exception handling. Since the arithmetic operations dictated by JVM [16] is different from that implemented in X86 CPU [11, 12]. We need to pay special attention to arithmetic operations. Section 4 discusses the translation of arithmetic operations. In section 5, we present a template-based optimization technique. Section 6 compares the performance of our system with that of Java interpreter and Just-In-Time (JIT) compiler. Appendix A includes the assembly code for the `GetFnPtr` macro, the `build_objmapclasstable` and `search_objmapclass table` procedures. Appendix B is a list of patterns for generating better assembly code.

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. The `GetFnPtr` macro, included in Appendix A, makes use

of that reference to invoke JNI functions [23, 15]. The reference to JVM is also used to invoke methods in the bytecode form, such as all methods in the standard packages in Java. Note that our translator allows mixed execution of assembly code and bytecode. JNI functions are the interface between the two forms of code.

To start up a JVM, use

```
res = JNI_CreateJavaVM(&jvm, &env, &vm_args);
```

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. Below is the X86 code for creating an instance.

```
GetFnPtr fntblptr, 6, fnptr ; FindClass
push offset index_2_clsname
push jnienv
call [fnptr]
mov index_2_cls, eax
push index_2_cls
pop eax
push eax
push eax
mov eax, 100
push eax
pop [argreversebuf+0*4]
pop jclsjobtmp
GetFnPtr fntblptr, 33, fnptr ; GetMethodID
push offset index_3_midsig
push offset index_3_midname
push jclsjobtmp
push jnienv
call [fnptr]
mov index_3_mid, eax
GetFnPtr fntblptr, 28, fnptr ; NewObject
push [argreversebuf+0*4]
push index_3_mid
```

```

push jclsjobtmp
push jnienv
call [fnptr]
add esp, 20
push eax
pop [aa_local_vars+1*4]

```

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 corresponding X86 code is as follows:

```
pop jclsjobtmp ; object reference
GetFnPtr fntblptr, 6, fnptr ; FindClass
push offset index_2_clsname
push jnienv
call [fnptr]
mov index_2_cls, eax
GetFnPtr fntblptr, 94, fnptr ; GetFieldID
push offset index_2_fidsig
push offset index_2_fidname
push index_2_cls
push jnienv
call [fnptr]
mov index_2_fid, eax
GetFnPtr fntblptr, 100, fnptr ; Get<Type>Field
push index_2_fid
push jclsjobtmp
push jnienv
call [fnptr]
push eax
. . . ; +1
pop [argreversebuf]
pop jclsjobtmp ; object reference
GetFnPtr fntblptr, 6, fnptr ; FindClass
push offset index_2_clsname
push jnienv
call [fnptr]
mov index_2_cls, eax
```

```

GetFnPtr fntblptr, 94, fnptr ; GetFieldID
push offset index_2_fidsig
push offset index_2_fidname
push index_2_cls
push jnienv
call [fnptr]
mov index_2_fid, eax
GetFnPtr fntblptr, 109, fnptr ; Set<Type>Field
push [argreversebuf]
push index_2_fid
push jclsjobtmp
push jnienv
call [fnptr]

```

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. The corre-

spending X86 assembly code is as follows:

```
push [aa_local_vars+1*4] ; aload.1 (variable a)
GetFnPtr fntblptr, 6, fnptr ; FindClass
push offset index_2_clsname
push jnienv
call [fnptr]
mov index_2_cls, eax
GetFnPtr fntblptr, 32, fnptr ; IsInstanceOf
pop eax
push index_2_cls
push eax
push jnienv
call [fnptr] ; result in eax
push eax
pop eax ; ifeq 15
cmp eax, 0
jz aa_15
```

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
```

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[16].

```

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, of which the code is included in the appendix.

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.

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

The `search_objmapclasstable` procedure, of which the code is included in the appendix, 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

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

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`. For instance, the Java source code `return -1` is translated into the two bytecode instructions:

```
iconst_m1 ; parameter in the stack
ireturn
```

which is translated into the following X86 assembly code:

```
mov eax, -1
push eax
push aa_return_addr
ret
```

3 Translating Exceptions

Java provides extensive supports for handling exceptions. Exceptions are modelled as objects (which belong to the class `java.lang.Throwable`) with the usual rules of inheritance in Java.

3.1 Throwing an Exception

The Java source code `throw new RuntimeException();` is translated into the following four bytecode instructions:

```
new #2 ; <Class RuntimeException>
dup
invokespecial #3 ; <Method RuntimeException()>
athrow
```

The first three instructions create a `RuntimeException` object and the last throws that object.

The `athrow` instruction is translated into calls to the JNI functions `GetObjectClass` and `ThrowNew`. The `GetObjectClass` function obtains the class of the exception object. The `ThrowNew` function informs JVM that an exception of the intended class has been thrown.

3.2 Catching an Exception

3.2.1 Building the Exception Table

If a method includes exception handlers, there will be an exception table following the method in the bytecode file. In the assembly program, we also need to build a corresponding exception table. An entry of the exception table has the following structure:

```
exceptionEntry struc
from dword ?
to dword ?
target dword ?
type dword ?
exceptionEntry ends
```

The `from` and `to` fields denote the range of the bytecode instructions that are in the `try` clause. The `target` field denotes the handler. The `type` field is the

`ClassId` of the exception that the handler will take care of. This `ClassId` is obtained by calling the JNI function `FindClass`. The assembly code for building an entry of the exception table is as follows:

```
mov esi, offset allexception
mov (exceptionEntry ptr [esi]).from, 2
mov (exceptionEntry ptr [esi]).to, 10
mov (exceptionEntry ptr [esi]).target, 10
push offset exception_0
push jnienv
GetFnPtr fntblptr, 6, fnptr ; FindClass
call [fnptr]
mov (exceptionEntry ptr [esi]).type, eax
```

3.2.2 Selecting an Exception Handler

In order to catch an exception, we need to detect when an exception has occurred. There are three places in the assembly code that we need to check the occurrences of exception: (1) immediately after an exception has been thrown, (2) upon returning from a method call, and (3) before performing division.

Immediately after an exception has been thrown, we add a call to the JNI function `ExceptionCheck` to catch that exception. That call serves to catch the thrown exception.

When a method calls another method, the calling method must check if an exception has occurred in the called method upon returning from the called method. The JNI function `ExceptionCheck` is invoked for this purpose.

In Java, JVM only processes the integer-division-by-zero exception automatically. Java does not check overflow and underflow unless a programmer explicitly adds Java statements for that purpose. Before, performing any division, assembly code—`cmp divisor, 0` and `je Exhandle`—for checking if the divisor is zero is added, which is followed by a jump to an exceptional

routine. In 32-bit division, we can easily check the divisor. In 64-bit division, we must check two 32-bit parts. Note that division-by-zero is not checked for floating-point divisions.

The JIN function `ExceptionCheck` returns 1 if an exception occurred; otherwise it returns 0. When `ExceptionCheck` returns 1, execution continues to the code labelled `Exhandle`, where a search of the exception table for a handler of the intended type is initiated.

The search first obtains the exception object from JVM by calling the JNI function `ExceptionOccurred`. Then, check if the instruction address where the exception occurred falls in the proper range depicted by the `from` and `to` fields of the `exceptionEntry`. If so, invoke the JNI function `IsInstanceOf` to check if the exception object is of the type handled by the handler. If both the range and type are matched, the address of the handler (that is, the `target` field) is returned. The assembly code for searching the exception table is as follows:

```
push jnienv
GetFnPtr fntblptr, 15, fnptr ; ExceptionOccurred
call [fnptr]
mov exobj, eax
. . .
.while edi < exnum
mov ebx, (exceptionEntry ptr [esi]).from
mov ecx, (exceptionEntry ptr [esi]).to
.if (ebx <= index) && (index < ecx)
mov edx, (exceptionEntry ptr [esi]).type
push edx
push exobj
push jnienv
GetFnPtr fntblptr, 32, fnptr ; IsInstanceOf
call [fnptr]
```

```

.if eax == 1
mov edx, (exceptionEntry ptr [esi]).target
push edx
GetFnPtr fntblptr, 17, fnptr
push jnienv
call [fnptr]
.break
.endif
.endif
add esi, type exceptionEntry
inc edi
.endw

```

After a proper exception handler is found, the JNI function `ExceptionClear` is invoked to clear the exception flag that was raised when the exception occurred. Execution then continues to the exception handler. (If no proper handler can be found, the Java compiler will generate a default handler which will raise the exception again, after the `finally` block is done. Thus, we may assume that there is always an exception handler in the bytecode. Catch clauses with `finally` blocks are a little simpler and are implemented similarly.)

3.2.3 Exception Handlers

Exception handlers are translated into assembly code in the same manner as normal code. At the end of the handler, we invoke the subroutine representing the `finally` block, if there is one such `finally` block, and then either `goto` the end of the `try` block (for the proper handler) or raises the exception again (for the default handler).

3.2.4 finally Blocks

A `finally` block in bytecode has the form of a subroutine. The handler is entered with the `jsr` instruction and exited with the `ret` instruction. The `jsr` instruction is translated into a sequence of three assembly instructions:

```
mov eax, 8
push eax
jmp aa_17
```

The instructions at the address `aa_17` will pop the item on the stack top (that is, the return address) into a local variable. The `ret` instruction depends on this return address. The `ret` instruction is translated into three assembly instructions:

```
mov eax, [aa_local_vars+3*4]
cmp eax, 8
je aa_8
```

4 Translating Ordinary Instructions

4.1 Operand Stack

The operand stack in Java virtual machine is implemented with the stack in X86 CPU. For instance, the instruction `iadd` is implemented with the following sequence of 4 X86 instructions:

```
pop eax
pop ebx
add eax, ebx
push eax
```

Other integer operations, such `isub`, `imul`, `idiv`, `irem`, `ineg`, `ishl`, `ishr`, `iushr`, `iand`, `ior`, `ixor`, are implemented similarly.

4.2 Floating-Point Operations

The floating-point unit in the X86 CPU supports a stack for floating-point operations. For instance, the JVM instruction `fadd` is implemented with the following sequence of 8 X86 instructions:

```
pop real4buf
fld real4buf
pop real4buf
fld real4buf
fadd
fstp real4buf
push real4buf
```

Here the variable `real4buf` is a 4-byte scratch variable defined in the X86 assembly program.

The JVM instruction `dadd` is similarly implemented with the following sequence of 11 X86 instructions:

```
pop dword ptr real8buf+4
pop dword ptr real8buf
fld real8buf
pop dword ptr real8buf+4
pop dword ptr real8buf
fld real8buf
fadd
fstp real8buf
push dword ptr real8buf
push dword ptr real8buf+4
```

Here the variable `real8buf` is a 8-byte scratch variable defined in the X86 assembly program.

Other floating-point operations, including `fsub`, `fmul`, `fdiv`, `fneg`, `dsub`, `dmul`, `ddiv`, `dneg`, are implemented similarly.

Note that the `frem` and `drem` instructions in JVM need special attention. In X86, which follows IEEE 754 standard, they are computed with a rounding division [13, 5, 12]. However, in JVM, they are defined with a truncating division. We use appropriate instruction sequences to implement the remainder operations. Our algorithm follows the one in [16].

Conversion of floating-point values to integer values also needs special attention. The conversion operation in X86 architecture follows the IEEE 754 standard [11]. On the other hand, the conversion defined in JVM [16] dictates special rules for the following five special values: `NaN` (*not-a-number*), positive and negative infinities, and largest and smallest values. We add code to convert these special values. For other normal values, we use the X86 conversion instruction `fistp` directly [12].

4.3 Long-Integer Operations

Long-integer operations are more difficult because X86 does not support them directly. We implemented the `long` integer instructions by adding four-byte integer values twice, of which the second addition is implemented with the `adc` (add with carry) instruction. Subtraction of long integers is implemented similarly with the `sbb` instruction.

In our original implementation, multiplications (and divisions) of `long` integers were implemented as a series of multiplications (and divisions, respectively) of integers. However, the performance of this implementation was worse than direct interpretation of JVM (without the help of JIT). In our revised implementation, multiplications and divisions of `long` integers are implemented with those of the `double` values with the necessary conversions between `long` values and `double` values. The performance of the revised implementation is comparable to that of direct interpretation of JVM (without the help of JIT).

4.4 Arrays

4.4.1 Simple Arrays

The bytecode instruction set includes three instructions for creating arrays. The `newarray` instruction constructs one-dimension arrays with the primitive types (`boolean`, `char`, `float`, `double`, `byte`, `short`, `int`, `long`). The `anewarray` instruction constructs a one-dimension array with reference type. The `multianewarray` instruction constructs multi-dimension arrays. The effect of `multianewarray` can be achieved with repeated use of `anewarray`.

In our translation, arrays are created inside the JVM. The Java source code `double[] ir = new double[10]`, which creates a one-dimensional array with 10 `double` elements, is translated into the following bytecode:

```
bipush 10
newarray double
astore_1
```

This sequence of bytecode instructions is implemented by a call to the JNI function `NewDoubleArray`:

```
GetFnPtr fntblptr, 182, fnptr ; NewDoubleArray
push jnienv
call [fnptr]
push eax
pop [aa_local_vars+1*4]
```

To access an element of the array, for instance, `ir[i]=16.23` is translated into the following bytecode:

```
aload_1 ; ir
iload_2 ; i
ldc2_w #2 ; double 16.23
dastore
```

This assignment to an element of a double array is implemented by a call to the JNI function `SetDoubleArrayRegion`:

```
push [aa_local_vars+1*4] ; ir
pop [argreversebuf+4] ; ir
push [aa_local_vars+2*4] ; i
pop [argreversebuf] ; i

push offset real8buf ; double 16.23
mov eax, 1
push eax
push [argreversebuf] ; i
push [argreversebuf+4] ; ir
GetFnPtr fntblptr, 214, fnptr ; SetDoubleArrayRegion
push jnienv
call [fnptr]
```

The length of an array may be obtained by the Java source code `ir.arraylength`. This expression is translated into the following bytecode:

```
aload_1 ; ir
arraylength
```

The bytecode instruction sequence is implemented by a call to the JNI function `GetArrayLength`:

```
push [aa_local_vars+1*4] ; ir
GetFnPtr fntblptr, 171, fnptr ; GetArrayLength
push jnienv
call [fnptr]
push eax ; push the length of the ir array onto stack
```

Other arrays of primitive types are implemented similarly.

4.4.2 Multi-Dimensional Arrays

For the `anewarray` and `multianewarray` instructions, the type of the elements is retrieved from the constant pool. Our implementation reserves a 20-element array `arangelist`, of which each element saves the ranges of indexes of a dimension. (Hence, our system can implement arrays of up to 20 dimensions.) We also reserves another 20-element array `aobjlist`, which holds the references to the created arrays:

```
arangelist dword 20 dup(?)
aobjlist dword 20 dup(?)
```

The Java statement `int[][][] threeD = new int[5][4][3]`, which creates a 3-dimensional `int` array, is translated into the following bytecode instructions:

```
iconst_5
iconst_4
iconst_3
multianewarray #2 dim #3 ; Class [[[I>
astore_1 ; stored in threeD
```

A 3-dimensional `int` array is implemented as a 1-dimensional array of which the element has type `[[I` (a 2-dimensional array). Similarly, a 2-dimensional array is implemented as a 1-dimensional array of which the element has type `[I`. Thus, we define two constants:

```
index_2_array0 db '[[I', 0
index_2_cls dword ?
```

The variable `index_2_cls` holds a reference to the type `[[I`. The assembly program first creates an object that represents a 3-dimensional array, whose element has type `[[I`:

```

mov eax, 5
push eax
pop [arangelist+0*4] ; index range of the first dimension
GetFnPtr fntblptr, 6, fnptr ; FindClass
push offset index_2_array0 ; '[[I'
push jnienv
call [fnptr] ; result in eax
mov index_2_cls, eax
GetFnPtr fntblptr, 172, fnptr ; NewObjectArray
mov eax, 0
push eax
push index_2_cls
mov eax, [arangelist+0*4]
push eax
push jnienv
call [fnptr] ; result in eax
mov [aobjlist+0*4], eax ; save the created ref in [aobjlist+0*4]

```

After creating the 3-dimensional 5-element array (whose element type is `[[I]`), the assembly program creates 5 2-dimensional 4-element array (whose element type is `[I]`) recursively.

To access an element of a multi-dimensional array, for instance, the Java assignment `threeD[i][j][k] = i + j` is compiled into the following byte-code instructions:

```

aload_1 ; threeD
iload_2 ; i
aaload ; threeD[i]
iload_3 ; j
aaload ; threeD[i][j]
iload 4 ; k
iload_2 ; i

```

```
    iload_3 ; j
    iadd ; i + j
    iastore
```

The `aaload` instruction makes use of the assembly procedure `update_arrayelementobj` to maintain the latest reference loaded from an element of an array into a `objmapclass` structure. The reason why we need `update_arrayelementobj` is that when the elements of an array are object references, the object references may change every time the JNI functions that implement the `aaload` instruction access the references. The assembly code looks roughly like

```
    push eax
    push offset aa_objmapclass
    push eax
    call update_arrayelementobj
```

4.5 Switch

The `switch` statement in Java is compiled into the `lookupswitch` or `tableswitch` [16]. The `lookupswitch` bytecode instruction is implemented with a series of `cmp eax, aChoice` and `je somewhere` instructions in the assembly code. The `tableswitch` bytecode instruction is implemented with a jump table in the assembly code.

5 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 [10, 14], 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. They are listed in the appendix of this paper.

6 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. The operating system is 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.

We first runs an empty loop 1,000,000,000 times. The execution time of the assembly program is roughly 1/5 of that of Java interpreter and is about

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

1.7 times that of JIT. JIT is better in this case because it fully utilizes the registers in the X86 CPU whereas our translator does not.

We then measure the execution time of arithmetic operations. Each test case is executed 100,000,000 times. The performance of the resulting assembly code on `int` computation and `long` additions and `long` subtractions are as good as expected, and if we make more template-based optimizations in the future, the performance will be close to the JIT. On the other hand, the performance of `long` multiplications and divisions is not as good as expected. The reason is that `long` multiplications and divisions, which are not supported directly in X86 CPU, are done with software (the *longhand multiplication* and *nonrestoring division* algorithms [18, 5]) in our translator. In the revised version of the translator, `long` multiplications and divisions are done with `double` operations with the necessary conversions. The performance of the revised version is comparable to that of the interpreter.

The performance of floating-point operations, which are done with FPU in X86, is better than that of the interpreter, but worse than that of JIT. This is because local variables are saved in the memory. Hence, the assembly code must get the value of the local variable through memory.

In the third part, we evaluated array performance. Three cases are tested: (1) A one-dimension `int` array with 10,000,000 elements is created. (2) A three-dimension `int` array with 60 elements is created. Each element is assigned an integral value. The assignment is executed 1,000 times. (3) A one-dimension `int` array with 3 elements is created for 1,000,000 times.

In the fourth part, we tested field access. An instance which has an `int` field and a class field is created. We then access each field 1,000,000 times.

Arrays and fields are implemented with JNI functions, which incur some overhead. A way to improve the performance for arrays and fields is to cache the results of JNI functions because many invocations of JNI functions are identical and produce the same results.

In the sixth part, we tested method invocation. An empty instance

method and an empty static method are called 1,000,000 times.

The performance of calling a static method in the resulting assembly code is comparable to that of JIT. The reason is that our translator can decide the actual method and generate efficient assembly code.

Calling an instance method only once in the assembly code is faster than the interpreter. However, when an instance method is invoked repeatedly, the interpreter may re-use the previously resolved instance method. On the other hand, the assembly code generated by our translation system always repeats the dynamic dispatching procedure whenever a method is called. This is so even if the same instance method is invoked. Therefore, the performance of the assembly program is comparable to that of the interpreter when an instance method is invoked 1,000,000 times. To improve the performance, our translator should generate a caching dynamic dispatching procedure.

Finally, we tested the Linpack[6] and Btest[7] programs. Btest is a benchmark for Java arrays, which include many `int` and `double` computations. The Linpack Benchmark performs an intensive numeric test to measure the floating point performance of computers. The performance of the resulting assembly code is close to that of the interpreter, and there is a large performance gap between the assembly code and the JIT. Because the Linpack and BTest have a lot of shuffling-around in array computations, the possible reason is the overhead of the JNI functions.

7 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] Ali-Reza Adl-Tabatabai, M. Cierniak, G.Y. Lueh, V.M. Parikh, and J.M. Stichnoth, "Fast and effective code generation in a just-in-time Java compiler," ACM SIGPLAN Notices, 33(5), 280-290, May 1998.
- [2] A. Azevedo, A. Nicolau, and J. Hummel, "Java Annotation-Aware Just-In-Time (AJIT) Compilation System," Proc. ACM 1999 Conference on Java Grande, 142-151, June 1999.
- [3] P. Bothner, "A Gcc-based Java Implementation," Proc. Compcon '97, 174-178, February 1997.
- [4] M. Cierniak, G.-Y. Lueh, J.M. Stichnoth, "Practicing JUDO: Java Under Dynamic Optimizations," Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, 35(5), 13-26, May 2000.
- [5] S.P. Dandamudi, Introduction to Assembly Language Programming, Springer, New York, 1998.
- [6] J. Dongarra, R. Wade, and P. McMahan, Linpack Benchmark – Java Version, available on the web at <http://www.netlib.org/benchmark/linpackjava/>, April 1996.
- [7] R. Grothmann, Java Benchmark, available on the web at <http://mathsrv.ku-eichstaett.de/MGF/homes/grothmann/java/bench/Bench.html>, 2003.
- [8] C.H.A. Hsieh, J.C. Gyllenhaal, and W.W. Hwu, "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results," Proc. 29th Ann. International Symp. Microarchitecture, Los Alamitos, CA, 90-97, 1996.

- [9] C.-H.A. Hsieh, M.T. Conte, T.L. Johnson, J.C. Gyllenhaal, W.-M.W. Hwu, "Compilers for Improved Java Performance," *Computer*, 30(6), 67-75, June 1997.
- [10] 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.
- [11] IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, Intel Corp., Mt. Prospect, IL, 2002.
- [12] IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference, Intel Corp., Mt. Prospect, IL, 2002.
- [13] K.R. Irvine, *Assembly Language for Intel-Based Computers*, 4th ed., Prentice-Hall, New Jersey, 2003.
- [14] H.D. Lambright, "Java Bytecode Optimizations," *Proc. Comcon '97*, 206-210, February 1997.
- [15] Sheng Liang, *The Java Native interface: Programmer's Guide and Specification*, Addison-Wesley, Reading, MA, 1999.
- [16] 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/>.
- [17] V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, and A. Yeryomin, "Overview of Excelsior JET, a high performance alternative to Java Virtual Machines," *Proc. 3rd International Workshop on Software and Performance*, 104-113, July 2002.

- [18] D.A. Patterson and J.L. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*, 2nd ed., Morgan Kaufmann Publishers, San Francisco, CA, 1998.
- [19] T.A. Proebsting, G. Townsend, P. Bridges, J.H. Hartman, T. Newsham, and S.A. Watterson, "Toba: Java for applications: A way ahead of time (WAT) compiler," *Proc. Usenix Association*, 41-54, 1997.
- [20] R. V. Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a Java Bytecode Optimization Framework," *Proc. 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1-11, November 1999.
- [21] B. Sarkar, 99% Java, Assembly and JNI, <http://www.amherst.edu/~tliron/jni/assembly.html>, 2002.
- [22] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-In-Time Compiler," *IBM Systems Journal*, 39(1), 175-193, 2000.
- [23] Sun, *Java Native Interface, Java™ 2 SDK, Standard Edition Documentation Version 1.4.0*, 2002.

Appendix A.

The macro `GetFnPtr`, taken from [21], returns a pointer to a JNI function. The variable `fntblptr` points to the table of all JNI functions. This pointer is obtained when the assembly code starts up a JVM. The assembly code of `GetFnPtr` is as follows:

```
GetFnPtr MACRO fntblptr, index, fnptr
mov eax, index
mov ebx, 4
mul ebx
mov ebx, fntblptr
add ebx, eax
mov eax, [ebx]
mov fnptr, eax
ENDM
```

The assembly procedure `build_objmapclasstable` is as follows:

```
build_objmapclasstable proc
.data
allobjmapclass_entry dword ?
objref dword ?
objref_entry dword ?
.code
pop ebp
pop objref_entry
pop objref
pop allobjmapclass_entry
mov ebx, dword ptr allobjmapclass_entry
insert_item:
cmp (objmapclass_table ptr [ebx]).jobj, -1
jne next_item
```

```

mov eax, dword ptr objref
mov (objmapclass_table ptr [ebx]).jobj, eax
mov eax, dword ptr objref_entry
mov (objmapclass_table ptr [ebx]).class, eax
jmp insert_ok
next_item:
add ebx, type objmapclass_table
jmp insert_item
insert_ok:
push ebp
ret
build_objmapclasstable endp

```

The assembly procedure `search_objmapclasstable` is as follows:

```

search_objmapclasstable proc
.data
s_allobjmapclass_entry dword ?
s_objref dword ?
.code
pop ebp
pop s_objref
pop s_allobjmapclass_entry
mov edx, s_objref
mov ebx, s_allobjmapclass_entry
s_search: ;check if matched ?
cmp (objmapclass_table ptr [ebx]).jobj, edx
jne s_nextitem
mov eax, (objmapclass_table ptr [ebx]).class
jmp s_finish
s_nextitem:
add ebx, type objmapclass_table

```

```

cmp (objmapclass_table ptr [ebx]).jobj, -1
je s_notfind
jmp s_search
s_notfind:
mov eax, -1
s_finish:
push ebp
ret
search_objmapclasstable endp

```

Appendix B. Here is the list of templates consisting of two bytecode instructions:

- The first bytecode instruction is one of `iload/iload_w/iload_[0/1/2/3]` and the second bytecode instruction is one of `ifeq/ifgt/iflt/ifle/ifne/ifge`. Similarly for the other patterns listed below.
- `aload/aload_w/aload_[0/1/2/3]`
`ifnull/ifnonnull`
- `bipush/sipush/iconst_m1/iconst_[0/1/2/3/4/5]`
`ifeq/ifgt/iflt/ifle/ifne/ifge`
- `bipush/sipush/iconst_m1/iconst_[0/1/2/3/4/5]`
`istore_[0/1/2/3]/istore/istore_w`
- `iload_[0/1/2/3]/iload/iload_w`
`istore_[0/1/2/3]/istore/istore_w`

Here is the list of templates consisting of three bytecode instructions:

- `iload/iload.w/iload_[0/1/2/3]`
`iload/iload.w/iload_[0/1/2/3]`
`if_icmpeq/if_icmpne/if_icmplt/if_icmpgt/if_icmple/if_icmpge`