Native Code Generation

Matthew Dwyer
324E Nichols Hall
dwyer@cis.ksu.edu
http://www.cis.ksu.edu/~dwyer
Compile Time

• JOOS programs are compiled into bytecode.

• This bytecode can be executed using:
  – an interpreter;
  – an Ahead-Of-Time (AOT) compiler; or
  – a Just-In-Time (JIT) compiler.

• Bytecode is compiled into native code.
Interpreters

• are easier to implement than compilers;
• can be very portable; but
• suffer an inherent inefficiency
pc = code.start;
while(true)
    {
        npc = pc + instruction_length(code[pc]);
        switch (opcode(code[pc]))
        {
            ILOAD_1: push(local[1]); break;
            ILOAD: push(local[code[pc+1]]); break;
            ISTORE: t = pop();
            local[code[pc+1]] = t; break;
            IADD: t1 = pop(); t2 = pop();
            push(t1 + t2); break;
            IFEQ: t = pop();
            if (t == 0) npc = code[pc+1]; break;
...
        }
        pc = npc;
    }
Ahead-of-Time Compilers

• translate the low-level intermediate form into native code;
• create all object files, which are then linked, and finally executed.

• This is not so useful for Java and JOOS:
  – method code is fetched as it is needed;
  – from across the internet; and
  – from multiple hosts with different native code sets.
Just-in-Time Compilers

- merge interpreting with traditional compilation;
- have the overall structure of an interpreter; but
- method code is handled differently.

- When a method is invoked for the first time:
  - the bytecode is fetched;
  - it is translated into native code; and
  - control is given to the newly generated native code.

- When a method is invoked subsequently:
  - control is simply given to the previously generated native code.
Features of a JIT Compiler

- it must be *fast*, because the compilation occurs at run-time;
  - Just-In-Time is really Just-Too-Late
- it does not generate optimized code;
- it does not compile every instruction into native code, but relies on the runtime library for complex instructions;
- it need not compile every method; and
- it may concurrently interpret and compile a method
  - Better-Late-Than-Never
Generating Native Code

instruction selection
  – choose the correct instructions based on the native code instruction set

memory modelling
  – decide where to store variables and how to allocate registers

method calls
  – determine calling conventions

branch handling
  – allocate branch targets
JVM to VirtualRISC

- map the Java local stack into registers and memory;
- do instruction selection on the fly;
- allocate registers on the fly; and
- allocate branch targets on the fly.

Several real JITs do this (e.g., Kaffe)
CodeGen Algorithm

- determine number of slots in frame:
  locals limit + stack limit + # temps
- find starts of basic blocks;
- find local stack height for each bytecode;
- emit prologue;
- emit native code for each bytecode; and
- fix up branches.
Naïve Approach

- each local and stack location is mapped to an offset in the native frame;
- each bytecode is translated into a series of native instructions, which
- constantly move locations between memory and registers.

- This is similar to the native code generated by a non-optimizing compiler.
public void foo() {
    int a, b, c;
    a = 1;
    b = 13;
    c = a + b;
}
Example : Generated Bytecode

```java
.method public foo()V
.limit locals 4
.limit stack 2
iconst_1 ; 1
istore_1 ; 0
ldc 13 ; 1
istore_2 ; 0
iload_1 ; 1
iload_2 ; 2
iadd ; 1
istore_3 ; 0
return ; 0
```

- compute frame size
  - \( 4 + 2 + 0 = 6 \);
- find stack height for each bytecode;
- emit prologue; and
- emit native code for each bytecode.
### Example: Frame Slot Assignments

<table>
<thead>
<tr>
<th>name</th>
<th>offset</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>[fp-32]</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>[fp-36]</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>[fp-40]</td>
</tr>
<tr>
<td>stack</td>
<td>0</td>
<td>[fp-44]</td>
</tr>
<tr>
<td>stack</td>
<td>1</td>
<td>[fp-48]</td>
</tr>
</tbody>
</table>
a = 1;  iconst_1
        mov 1, R1
        store R1, [fp-44]
        istore_1  ld [fp-44], R1
                                                    st R1, [fp-32]

b = 13;  ldc 13  mov 13, R1
        store R1, [fp-44]
        istore_2  ld [fp-44], R1
                                                    st R1, [fp-36]

c = a + b;  iload_1  ld [fp-32], R1
                                                        st R1, [fp-44]
        iload_2  ld [fp-36], R1
                                                        st R1, [fp-48]
iadd  ld [fp-48], R1
                                                        ld [fp-44], R2
                                                        add R2, R1, R1
                                                        st R1, [fp-44]
        istore_3  ld [fp-44], R1
                                                        st R1, [fp-40]

return  restore
        ret
Naïve Code is Slow

• Many unnecessary loads and stores
  – Which are the most expensive operations
• We would like to replace load/store operations with register operations
  – Requires that we assign registers to JVM local stack locations
Naïve Code is Slow

c = a + b; iload_1 ld [fp-32],R1
           st R1,[fp-44]
iload_2 ld [fp-36],R1
           st R1,[fp-48]
iadd ld [fp-48],R1
           ld [fp-44],R2
           add R2,R1,R1
           st R1,[fp-44]
istore_3 ld [fp-44],R1
           st R1,[fp-40]

Becomes

c = a + b; iload_1 ld [fp-32],R1
iload_2 ld [fp-36],R2
iadd add R1,R2,R1
istore_3 st R1,[fp-40]
Register Allocation

A fixed register allocation scheme:
- assign $m$ registers to the first $m$ locals;
- assign $n$ registers to the first $n$ stack locations;
- assign $k$ scratch registers; and
- spill remaining locals and locations into memory.

Example for 6 registers (m=n=k=2):

<table>
<thead>
<tr>
<th>name</th>
<th>offset</th>
<th>location</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>[fp-40]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>stack</td>
<td>stack 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>scratch</td>
<td>0</td>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>scratch</td>
<td>1</td>
<td>R6</td>
<td></td>
</tr>
</tbody>
</table>
Register Allocation Improves Code

\[
\begin{align*}
a &= 1; & \text{iconst}_1 & \text{mov} 1, R3 \\
& & \text{istore}_1 & \text{mov} R3, R1 \\
b &= 13; & \text{ldc} 13 & \text{mov} 13, R3 \\
& & \text{istore}_2 & \text{mov} R3, R2 \\
c &= a + b; & \text{i10ad} & \text{mov} R1, R3 \\
& & \text{i10ad} & \text{mov} R2, R4 \\
& & \text{iadd} & \text{add} R3, R4, R3 \\
& & \text{istore}_3 & \text{st} R3, [fp-40] \\
& & \text{return} & \text{restore} \\
& & & \text{ret}
\end{align*}
\]
Register Allocation Improves Code

• This works quite well if:
  – the architecture has a large register set;
  – the stack is small most of the time; and
  – the first locals are used most frequently.
Summary of Fixed Register Allocation

• registers are allocated once; and
• the allocation does not change within a method.

• Advantages:
  – it's simple to do the allocation; and
  – no problems with different control flow paths.

• Disadvantages:
  – assumes the first locals and stack locations are most important; and
  – may waste registers within a region of a method.
The basic block register allocation scheme:

– assign frame slots to registers on demand within a basic block; and

– update descriptors at each bytecode.

The descriptor maps a slot to an element of the set

\[ \mu \in \{ P_i, \mu \in \mu \& P_i \} \]
Basic Block Register Allocation

For our example:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>R2</td>
</tr>
<tr>
<td>b</td>
<td>mem</td>
</tr>
<tr>
<td>c</td>
<td>mem&amp;R4</td>
</tr>
<tr>
<td>s_0</td>
<td>R1</td>
</tr>
<tr>
<td>s_1</td>
<td></td>
</tr>
</tbody>
</table>

We also keep the inverse register map:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>s_0</td>
</tr>
<tr>
<td>R2</td>
<td>a</td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>c</td>
</tr>
<tr>
<td>R5</td>
<td></td>
</tr>
</tbody>
</table>
Treating Control Flow

At the beginning of a block, all slots are in memory.

Basic blocks are merged by control paths:

```
\[
\begin{array}{|c|c|}
\hline
a & R1 \\
\hline
b & R2 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
a & R3 \\
\hline
b & R4 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
a & ? \\
\hline
b & ? \\
\hline
\end{array}
\]```
Treating Control Flow

Registers must be spilled after basic blocks:

- st R1, [fp-32]
- st R2, [fp-36]
- st R3, [fp-32]
- st R4, [fp-36]
Example: BB Register Allocation

```
save sp, -136, sp

iconst_1  mov 1, R1

istore_1  mov R1, R2
```
Example: BB Register Allocation

```
ldc 13  mov 13,R1
R1    s.0
R2    a
R3    _
R4    _
R5    _

istore_2 mov R1,R3
R1    _
R2    a
R3    b
R4    _
R5    _

iload_1 mov R2,R1
R1    s.0
R2    a
R3    b
R4    _
R5    _
```

```plaintext
  a  R2
  b  mem
  c  mem
  s.0 R1
  s.1 _
  a  R2
  b  R3
  c  mem
  s.0 _
  s.1 _
  a  R2
  b  R3
  c  mem
  s.0 R1
  s.1 _
```
Example: BB Register Allocation

\[
\begin{array}{c|c|c}
\text{iload}_2 & \text{mov R3,R4} \\
\hline
R1 & s_0 & a \\
R2 & a & b \\
R3 & b & c \\
R4 & s_1 & s_0 \\
R5 & _ & s_1 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{iadd} & \text{add R1,R4,R1} \\
\hline
R1 & s_0 & a \\
R2 & a & b \\
R3 & b & c \\
R4 & _ & s_0 \\
R5 & _ & s_1 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{istore}_3 & \text{mov R1,R4} \\
\hline
R1 & _ & a \\
R2 & a & b \\
R3 & b & c \\
R4 & c & s_0 \\
R5 & _ & s_1 \\
\end{array}
\]
Example: BB Register Allocation

```
st R2,[fp-32]
st R3,[fp-36]
st R4,[fp-40]
return
restore
ret
```

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>mem</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>mem</td>
</tr>
<tr>
<td></td>
<td>s_0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>s_1</td>
<td></td>
</tr>
</tbody>
</table>
But if we add the statement:

\[ c = c \times c + c; \]

then the fixed scheme generates:

```
ld [fp-40], R3
ld [fp-40], R4
mul R3, R4, R3
ld [fp-40], R4
add R3, R4, R3
st R3, [fp-40]
```

whereas the basic block scheme continues:

```
mul R4, R4, R1
add R1, R4, R1
st R1, R4
```

which is vastly better.
Summary of BB Register Allocation

• registers are allocated on demand; and
• slots are kept in registers within a basic block.

• Advantages:
  – registers are not wasted on unused slots; and
  – less spill code within a basic block.

• Disadvantages:
  – much more complex than the fixed register allocation scheme;
  – registers must be spilled at the end of a basic block; and
  – we may spill locals that are never needed.
Further Optimization

If we skip modeling the local stack

save sp,-136,sp  save sp,-136,sp

mov 1,R1       mov 1,R2
mov R1,R2
mov R1,R2
mov R1,R3
mov R1,R3
mov R2,R1
mov R3,R4
add R1,R4,R1       add R2,R3,R1
st R1,[fp-40]       st R1,[fp-40]
restore          restore
ret              ret
Unfortunately, this cannot be done safely on the fly by a peephole optimizer. The optimization:

\[
\text{mov } 1, \text{R3} \quad \text{mov } 1, \text{R1} \\
\text{mov } \text{R3}, \text{R1}
\]

is unsound if \( \text{R3} \) is used in a later instruction:

\[
\text{mov } 1, \text{R3} \quad \text{mov } 1, \text{R1} \\
\text{mov } \text{R3}, \text{R1} \\
\ldots \\
\text{mov } \text{R3}, \text{R4} \quad \text{mov } \text{R3}, \text{R4}
\]

Such optimizations require \text{dataflow analysis}.
Method Invocation

Invoking methods in bytecode:
– evaluate each argument leaving results on the stack; and
– emit `invokevirtual` instruction.

Invoking methods in native code:
– call library routine `soft_get_method_code` to perform the method lookup;
– generate code to load arguments into registers; and
– branch to the resolved address.
Example

Consider a method invocation

```
c = t.foo(a, b);
```

where the memory map is:

<table>
<thead>
<tr>
<th>name</th>
<th>offset</th>
<th>location</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>[fp-60]</td>
<td>R3</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>[fp-56]</td>
<td>R4</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>[fp-52]</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>4</td>
<td>[fp-48]</td>
<td>R2</td>
</tr>
<tr>
<td>stack</td>
<td>0</td>
<td>[fp-36]</td>
<td>R1</td>
</tr>
<tr>
<td>stack</td>
<td>1</td>
<td>[fp-40]</td>
<td>R5</td>
</tr>
<tr>
<td>stack</td>
<td>2</td>
<td>[fp-44]</td>
<td>R6</td>
</tr>
<tr>
<td>scratch</td>
<td>0</td>
<td>[fp-32]</td>
<td>R7</td>
</tr>
<tr>
<td>scratch</td>
<td>1</td>
<td>[fp-28]</td>
<td>R8</td>
</tr>
</tbody>
</table>
aload_4
iload_1
iload_2
invokevirtual foo  // soft call to get address
ld R7,[R2+4]
ld R8,[R7+52]
// spill all registers
st R3,[fp-60]
st R4,[fp-56]
st R2,[fp-48]
st R6,[fp-44]
st R5,[fp-40]
st R1,[fp-36]
st R7,[fp-32]
st R8,[fp-28]
// make call
mov R8,R0
call soft_get_method_code  // result in R0
// put args in R2, R1, and R0
ld R2,[fp-44]   \ R2 := stack_2
ld R1,[fp-40]   \ R1 := stack_1
st R0,[fp-32]   \ spill result
ld R0,[fp-36]   \ R0 := stack\_0
ld R4,[fp-32]   \ reload result
jmp [R4]   \ call method
Inefficient

• This is long and costly; and
• The lack of *dataflow analysis* causes massive spills within the basic blocks.
Handling Branches

• the only problem is that the target address is not known;
• assemblers normally handle this; but
• the JIT compiler produces binary code directly in memory.

Generating native code:

```java
if (a < b)  iload_1  ld R1,[fp-44]
  iload_2  ld R2,[fp-48]
  if_icmpge 17  sub R1,R2,R3
  bge ??
```

How to compute the branch targets:
– previously encountered branch targets are already known;
– keep unresolved branches in a table; and
– patch targets when the bytecode is eventually reached.