Virtual Machines

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

Abstract Syntax Trees

Virtual Machine Code

Native binary code

P-code, JVM

pentium, itanium
Compiling to VM Code

• Example:
  – gcc translates into RTL, optimizes RTL, and then compiles RTL into native code.

• Advantages:
  – exposes many details of the underlying architecture; and
  – facilitates production of code generators for many target architectures.

• Disadvantage:
  – a code generator must be built for each target architecture.
Interpreting VM Code

• Examples:
  – P-code for early Pascal interpreters;
  – Postscript for display devices; and
  – Java bytecode for the Java Virtual Machine.

• Advantages:
  – easy to generate the code;
  – the code is architecture independent; and
  – bytecode can be more compact.

• Disadvantage:
  – poor performance due to interpretative overhead (typically 5-20 times slower).
A Model RISC Machine

• VirtualRISC is a simple RISC machine with:
  – memory;
  – registers;
  – condition codes; and
  – execution unit.

• In this model we ignore:
  – caches;
  – pipelines;
  – branch prediction units; and
  – advanced features.
VirtualRISC Memory

• a stack
  – used for function call frames;
• a heap
  – used for dynamically allocated memory;
• a global pool
  – used to store global variables; and
• a code segment
  – used to store VirtualRISC instructions.
VirtualRISC Registers

- unbounded number of general purpose registers;
- the stack pointer \( (sp) \) which points to the top of the stack;
- the frame pointer \( (fp) \) which points to the current stack frame; and
- the program counter \( (pc) \) which points to the current instruction.
VirtualRISC Condition Codes

- stores the result of last instruction that can set condition codes (used for branching).
VirtualRISC Execution Unit

- reads the VirtualRISC instruction at the current $pc$, decodes the instruction and executes it;
- this may change the state of the machine (memory, registers, condition codes);
- the $pc$ is automatically incremented after executing an instruction; but
- function calls and branches explicitly change the $pc$. 
Memory/Register Instructions

\begin{align*}
\text{st } & R_i, [R_j] & & [R_j] := R_i \\
\text{st } & R_i, [R_j+C] & & [R_j+C] := R_i \\
\text{ld } & [R_i], R_j & & R_j := [R_i] \\
\text{ld } & [R_i+C], R_j & & R_j := [R_i+C]
\end{align*}
## Register/Register Instructions

- **mov Ri,Rj**  
  \[ Rj := Ri \]
- **add Ri,Rj,Rk**  
  \[ Rk := Ri + Rj \]
- **sub Ri,Rj,Rk**  
  \[ Rk := Ri - Rj \]
- **mul Ri,Rj,Rk**  
  \[ Rk := Ri \times Rj \]
- **div Ri,Rj,Rk**  
  \[ Rk := Ri \div Rj \]

Constants may be used in place of register values:

- **mov 5,R1**
Condition Instructions

Instructions that set the condition codes:
- `cmp Ri, Rj`

Instructions to branch:
- `b L`
- `bg L`
- `bge L`
- `bl L`
- `ble L`
- `bne L`

To express: *if R1 <= 9 goto L1*
We code: `cmp R1, 9`
- `ble L1`
Call Related Functions

save sp, -C, sp  

save registers, allocating $C$ bytes on the stack

call L

R15:=pc; pc:=L

restore

restore registers

ret

pc:=R15+8

nop

do nothing
local variables

space from 
alloca()

scratch space 
for register 
spills, temps

outgoing 
params

space to store 
register params 
from callee

space to save 
register window

Previous Frame

fp (old sp)

[fp-offset]

Current Frame

[sp+offset]
Stack Frames

- stores function activations;
- $sp$ and $fp$ point to stack frames;
- when a function is called a new stack frame is created:
  $$\text{push } fp; \; fp := sp; \; sp := sp + C;$$
- when a function returns, the top stack frame is popped:
  $$sp := fp; \; fp = \text{pop};$$
- local variables are stored relative to $fp$
- the figure shows additional features of the SPARC architecture.
Example C Code

```c
int fact(int n) {
    int i, sum;
    sum = 1;
    i = 2;
    while (i <= n){
        sum = sum * i;
        i = i + 1;
    }
    return sum;
}
```
Example VirtualRISC Code

_fact:
    save sp,-112,sp  // save stack frame
    st R0,[fp+68]    // save arg n in caller frame
    mov 1,R0         // R0 := 1
    st R0,[fp-16]    // sum is in [fp-16]
    mov 2,R0         // RO := 2
    st RO,[%fp-12]   // i is in [fp-12]
L3:
    ld [fp-12],R0    // load i into R0
    ld [fp+68],R1    // load n into R1
    cmp R0,R1        // compare R0 to R1
    ble L5           // if R0 <= R1 goto L5
    b L4             // goto L4
Example VirtualRISC Code

L5:

ld [fp-16],R0    // load sum into R0
ld [fp-12],R1    // load i into R1
mul R0,R1,R0     // R0 := R0 * R1
st R0,[fp-16]    // store R0 into sum
ld [fp-12],R0    // load i into R0
add R0,1,R1      // R1 := R0 + 1
st R1,[fp-12]    // store R1 into i
b L3             // goto L3

L4:

ld [fp-16],R0    // put return value into R0
restore          // restore register window
ret              // return from function
Java Virtual Machine

• memory;
• registers;
• condition codes; and
• execution unit.
JVM Memory

- a stack
  - used for function call frames;
- a heap
  - used for dynamically allocated memory;
- a constant pool
  - used for constant data that can be shared; and
- a code segment
  - used to store JVM instructions of currently loaded class files.
JVM Registers

- no general purpose registers;
- the stack pointer ($sp$) which points to the top of the stack;
- the local stack pointer ($lsp$) which points to a location in the current stack frame; and
- the program counter ($pc$) which points to the current instruction.
JVM Condition Codes

- stores the result of last instruction that can set condition codes (used for branching).
JVM Execution Unit

• reads the Java Virtual Machine instruction at the current \( \text{pc} \), decodes the instruction and executes it;
• this may change the state of the machine (memory, registers, condition codes);
• the \( \text{pc} \) is automatically incremented after executing an instruction; but
• method calls and branches explicitly change the \( \text{pc} \).
JVM Stack Frames

• Have space for:
  – a reference to the current object \(\text{this}\);  
  – the method arguments;  
  – the local variables; and  
  – a local stack used for intermediate results.

• The number of local slots and the maximum size of the local stack are fixed at compile-time.
Java Compilation

- Java compilers translate source code to class files.
- Class files include the bytecode instructions for each method.

```
Java Compiler
```

```text
a.java  javac  a.class
```

- Magic number
- Version number
- Constant pool
- Access flags
- this class
- super class
- Interfaces
- Fields
- Methods
- Attributes
Example Java Method

```java
public int Abs(int x) {
    if (x < 0)
        return(x * -1);
    else
        return(x);
}
```
Example JVM Bytecodes

.method public Abs(I)I // one int argument, returns an int
.limit stack 2     // has stack with 2 locations
.limit locals 2     // has space for 2 locals

// --locals--  --stack---
iload_1              // [ o -3 ]     [ * * ]
ifge Labell1
iload_1              // [ o -3 ]     [ -3 * ]
iconst_m1            // [ o -3 ]     [ -3 -1 ]
imul                // [ o -3 ]     [  3 * ]
ireturn              // [ o -3 ]     [ * * ]
Labell1:              
iload_1
ireturn
.ireturn
.end method
Bytecode Interpreter

```java
pc = code.start;
while(true)
    {  npc = pc + inst_length(code[pc]);
        switch (opcode(code[pc]))
            {  ILOAD_1: push(local[1]);
                break;

                ILOAD:   push(local[code[pc+1]]);
                break;

                ... 
            }
    }
    pc = npc;
}
Bytecode Interpreter

```java
pc = code.start;
while(true) {
    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;
}
```
JVM Arithmetic Operators

`ineg`   `[...:i]    -> [...:-i]`
`iadd`   `[...:i1:i2] -> [...:i1+i2]`
`isub`   `[...:i1:i2] -> [...:i1-i2]`
`imul`   `[...:i1:i2] -> [...:i1*i2]`
`idiv`   `[...:i1:i2] -> [...:i1/i2]`
`irem`   `[...:i1:t2] -> [...:i1\%i2]`
`iinc k a` `[...]  -> [...]`

`local[k]=local[k]+a`
JVM Branch Operations

goto L         [...]  ->  [...]  
branch always

ifeq L         [...]  ->  [...]  
branch if  i  ==  0

ifne L         [...]  ->  [...]  
branch if  i  !=  0

ifnull L       [...]  ->  [...]  
branch if  o  ==  null

ifnonnull L    [...]  ->  [...]  
branch if  o  !=  null
More Branches

\[
\text{if\_icmpeq} \ L \ [\ldots:i1:i2] \rightarrow [\ldots] \\
\text{branch if } i1 \ == \ i2
\]

\[
\text{if\_icmpne} \ L \ [\ldots:i1:i2] \rightarrow [\ldots] \\
\text{branch if } i1 \ != \ i2
\]

\[
\text{if\_icmpgt} \ L \ [\ldots:i1:i2] \rightarrow [\ldots] \\
\text{branch if } i1 \ > \ i2
\]

\[
\text{if\_icmplt} \ L \ [\ldots:i1:i2] \rightarrow [\ldots] \\
\text{branch if } i1 \ < \ i2
\]
More Branches

if_icmple L  [...:i1:i2] -> [...]
   branch if  i1 <= i2
if_icmpge L  [...:i1:i2] -> [...]
   branch if  i1 >= i2
if_acmpeq L  [...:o1:o2] -> [...]
   branch if  o1 == o2
if_acmpne L  [...:o1:o2] -> [...]
   branch if  o1 != o2
## Loading Constants

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iconst_0</code></td>
<td>[...] -&gt; [...:0]</td>
</tr>
<tr>
<td><code>iconst_1</code></td>
<td>[...] -&gt; [...:1]</td>
</tr>
<tr>
<td><code>iconst_2</code></td>
<td>[...] -&gt; [...:2]</td>
</tr>
<tr>
<td><code>iconst_3</code></td>
<td>[...] -&gt; [...:3]</td>
</tr>
<tr>
<td><code>iconst_4</code></td>
<td>[...] -&gt; [...:4]</td>
</tr>
<tr>
<td><code>iconst_5</code></td>
<td>[...] -&gt; [...:5]</td>
</tr>
<tr>
<td><code>aconst_null</code></td>
<td>[...] -&gt; [...:null]</td>
</tr>
<tr>
<td><code>ldc i</code></td>
<td>[...] -&gt; [...:i]</td>
</tr>
<tr>
<td><code>ldc s</code></td>
<td>[...] -&gt; [...:String(s)]</td>
</tr>
</tbody>
</table>
Memory Access

`iload k` \[\ldots\] \(\rightarrow\) \[\ldots::\text{local}[k]\]

`istore k` \[\ldots::i\] \(\rightarrow\) \[\ldots\]

local\[k\]=i

`aload k` \[\ldots\] \(\rightarrow\) \[\ldots::\text{local}[k]\]

`astore k` \[\ldots::o\] \(\rightarrow\) \[\ldots\]

local\[k\]=o

`getfield f sig` \[\ldots::o\] \(\rightarrow\) \[\ldots::o.f\]

`putfield f sig` \[\ldots::o:v\] \(\rightarrow\) \[\ldots\]

o.f=v
<table>
<thead>
<tr>
<th>Operation</th>
<th>Old Stack</th>
<th>New Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>dup</td>
<td>[...:v1]</td>
<td>[...:v1:v1]</td>
</tr>
<tr>
<td>pop</td>
<td>[...:v1]</td>
<td>[...: ]</td>
</tr>
<tr>
<td>swap</td>
<td>[...:v1:v2]</td>
<td>[...:v2:v1]</td>
</tr>
<tr>
<td>nop</td>
<td>[...: ]</td>
<td>[...: ]</td>
</tr>
</tbody>
</table>
new C  [...] -> [...:o]

instance_of C  [...:o] -> [...:i]
    if (o==null) i=0
else i=(C<=type(o))

checkcast C  [...:o] -> [...:o]
    if (o!=null &amp;&amp; !C<=type(o))
    throw ClassCastException
invokevirtual m sig [...
:o:a_1:...:a_n] -> [...]

entry=lookup(m,sig,o.methods);
block=select(entry,type(o));
push frame of size block.locals+block.stacksize;
local[0]=o;
local[1]=a_1;
...
local[n]=a_n;
pc=block.code;
Method Operations

`invokenonvirtual m sig
    [...:o:a_1:...:a_n] -> [...]

block=lookup(m,sig,o.methods);
push stack frame of size
    block.locals+block.stacksize;
local[0]=o;
local[1]=a_1;
...
local[n]=a_n;
pc=block.code;`
Method Operations

ireturn  [...:i] -> [...]
return i and pop stack frame

areturn  [...:o] -> [...]
return o and pop stack frame

return  [...] -> [...]
pop stack frame
public boolean member(Object item) {
    if (first.equals(item))
        return true;
    else if (rest == null)
        return false;
    else
        return rest.member(item);
}
.method public member(Ljava/lang/Object;)Z
.limit locals 2    // local[0] = o
    // local[1] = item
.limit stack 2     // initial stack [ * * ]
aload_0            // [ o * ]
getfield Cons/first Ljava/lang/Object;
    // [ o.first *]
aload_1            // [ o.first item]
invokevirtual     
    java/lang/Object/equals(Ljava/lang/Object;)Z
    // [bool *]
Corresponding Bytecode

ifeq else_0  // [  *  *  ]
iconst_1    // [  1  *  ]
ireturn     // [  *  *  ]
else_1:
aload_0     // [  o  *  ]
getfield Cons/rest LCons;  // [  o.rest  *  ]
aconst_null  // [  o.rest null]
if_acmpne else_2  // [  *  *  ]
iconst_0     // [  0  *  ]
ireturn      // [  *  *  ]
else_2:
aload_0 // [ o * ]
getfield Cons/rest LCons; // [ o.rest * ]
aload_1 // [ o.rest item ]
Invokevirtual Cons/member(Ljava/lang/Object;)Z // [ bool * ]
ireturn // [ * * ]
.end method
Bytecode Verification

- bytecode cannot be trusted to be well-formed and well-behaved;
- before executing any bytecode that is received over the network, it should be verified;
- verification is performed partly at class loading time, and partly at run-time; and
- at load time, dataflow analysis is used to approximate the number and type of values in locals and on the stack.
Properties of Verified Bytecode

• each instruction must be executed with the correct number and types of arguments on the stack, and in locals (on all execution paths);
• at any program point, the stack is the same size along all execution paths; and
• no local variable can be accessed before it has been assigned a value.
Interpreting Java

• when a method is invoked, a classloader finds the correct class and checks that it contains an appropriate method;

• if the method has not yet been loaded, then it may be verified (remote classes);

• after loading and verification, the method body is interpreted; or

• the bytecode for the method is translated to native code (only for the first invocation).
How we will use JVM and VirtualRISC

• Future use of Java bytecode:
  – the JOOS compiler will produce Java bytecode in Jasmin format; and
  – the JOOS peephole optimizer transforms bytecode into more efficient bytecode.

• Future use of VirtualRISC:
  – Java bytecode can be converted into machine code at run-time using a JIT (Just-In-Time) compiler;
  – we will study some examples of converting Java bytecode into a language similar to VirtualRISC;
  – we will study some simple, standard optimizations on VirtualRISC.