Advanced Models and Programs

Bytecode and real machine code

Peter Sestoft
Friday 2010-02-19*

The Java Virtual Machine (JVM)

- Well-defined stack-oriented bytecode
- Special instructions to support OO, arrays, ...
- Metadata describing classes, fields, methods
Example Java program

class Node extends Object {
    Node next;
    Node prev;
    int item;
}

class LinkedList extends Object {
    Node first, last;
    void addLast(int item) {
        Node node = new Node();
        node.item = item;
        if (this.last == null) {
            this.first = node;
            this.last = node;
        } else {
            this.last.next = node;
            node.prev = this.last;
            this.last = node;
        }
    }
    void printForwards() {
    }
    void printBackwards() {
    }
}

The JVM class file format

```
header
    LinkedList extends Object
constant pool
    #1 Object.<init>()
    #2 class Node
    #3 Node.<init>()
    #4 int Node.item
    #5 Node LinkedList.last
    #6 Node LinkedList.first
    #7 Node Node.next
    #8 Node Node.prev
    #9 void Out.println()
fld
    first (#6)
    last (#5)
methods
    <init>()
    void addLast(int)
    void printForwards()
    void printBackwards()
```
Some JVM bytecode instructions

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>push constant</td>
<td>iconst, ldc, aconst_null, ...</td>
</tr>
<tr>
<td>arithmetic</td>
<td>iadd, isub, imul, idiv, irem, ineg, iinc, fadd, ...</td>
</tr>
<tr>
<td>load local variable</td>
<td>iload, aload, fload, ...</td>
</tr>
<tr>
<td>store local variable</td>
<td>istore, astore, fstore, ...</td>
</tr>
<tr>
<td>load array element</td>
<td>ilaload, baload, aaload, ...</td>
</tr>
<tr>
<td>stack manipulation</td>
<td>swap, pop, dup, dup_x1, dup_x2, ...</td>
</tr>
<tr>
<td>load field</td>
<td>getfield, getstatic</td>
</tr>
<tr>
<td>method call</td>
<td>invokevirtual, invokevirtual, invokespecial</td>
</tr>
<tr>
<td>method return</td>
<td>return, ireturn, areturn, freturn, ...</td>
</tr>
<tr>
<td>unconditional jump</td>
<td>goto</td>
</tr>
<tr>
<td>conditional jump</td>
<td>ifeq, ifne, iflt, ifle, ...; if_icmpeq, if_icmpne, ...</td>
</tr>
<tr>
<td>object-related</td>
<td>new, instanceof, checkcast</td>
</tr>
</tbody>
</table>

Type prefixes: i=int, a=object, f=float, d=double, s=short, b=byte, ...

JVM bytecode verification

The JVM bytecode is *statically verified* before execution:

- An instruction must work on stack operands and local variables of the correct type
- A method must use no more local variables and no more local stack positions than it claims to
- For every point in the bytecode, the local stack has the same depth whenever that point is reached
- A method must throw no more exceptions than it admits to
- The execution of a method must end with a return or throw instruction, not `fall off the end`
- Execution must not use one half of a two-word value (e.g. a long) as a one-word value (int)
Additional JVM runtime checks

- Array-bounds checks
- Array assignment checks: Can store only subtypes of A into an A-array
- Null-reference check (a reference is null or points to an object or array, because no pointer arithmetics)
- Checked casts: Cannot make arbitrary conversions between object classes
- Memory allocation succeeds or throws exception
- No manual memory deallocation or reuse

- Bottom line: A JVM program cannot read or overwrite arbitrary memory
- Better debugging, better security
- No buffer overflow attacks, worms, etc as in C/C++

Example JVM runtime state

```java
void m() {
    LinkedList lst = new LinkedList();
    lst.addLast(5);
    lst.addLast(7);
    Node node = lst.first;
}
```

![Example JVM runtime state diagram](image-url)
The .NET Common Language Infrastructure (CLI, CLR)

- Much the same philosophy and design as JVM, but with some improvements:
  - Standardized bytecode assembly format
  - Better versioning, strongnames, ...
  - Designed as target for multiple source languages (C#, VB.NET, JScript, Eiffel, F#, Python, Ruby, ...)
  - User-defined value types (structs)
  - Tail calls to support functional languages
  - True generic types in bytecode: safer, more efficient, and more complex...

- The .exe file = stub + bytecode
- Standardized as Ecma-335

Some .NET CLI bytecode instructions

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>push constant</td>
<td>ldc.i4, ldc.r8, ldnull, ldstr, ldtoken</td>
</tr>
<tr>
<td>arithmetic</td>
<td>add, sub, mul, div, rem, neg; add.ovf, sub.ovf, ...</td>
</tr>
<tr>
<td>load local variable</td>
<td>ldloc, ldarg</td>
</tr>
<tr>
<td>store local variable</td>
<td>stloc, starg</td>
</tr>
<tr>
<td>load array element</td>
<td>ldelem.i1, ldelem.i2, ldelem.i4, ldelem.r8</td>
</tr>
<tr>
<td>stack manipulation</td>
<td>pop, dup</td>
</tr>
<tr>
<td>load field</td>
<td>ldfld, ldstfld</td>
</tr>
<tr>
<td>method call</td>
<td>call, calli, callvirt</td>
</tr>
<tr>
<td>method return</td>
<td>ret</td>
</tr>
<tr>
<td>unconditional jump</td>
<td>br</td>
</tr>
<tr>
<td>conditional jump</td>
<td>brfalse, brtrue; beq, bge, bgt, ble, blt, ...; bge.un ...</td>
</tr>
<tr>
<td>object-related</td>
<td>newobj, isinst, castclass</td>
</tr>
</tbody>
</table>

Type suffixes: i1=byte, i2=short, i4=int, i8=long, r4=float, r8=double
Canonical compilation?

• Consider the Java/C#/C program:

```java
static void Main(string[] args) {
    int n = int.Parse(args[0]);
    int y;
    y = 1889;
    while (y < n) {
        y = y + 1;
        if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0)
            InOut.PrintI(y);
    }
    InOut.PrintC(10);
}
```

• Let us compile it twice:
  – as Java, to JVM bytecode
  – as C#, to CLR bytecode
Ten-minute exercise

- On a printout of the preceding slide
  - Draw arrows to indicate where jumps go
  - Draw blocks around the bytecode segments corresponding to fragments of the Java program

No generic types in JVM bytecode

- Type parameter T gets erased to Object
- No type at runtime to represent T

```java
class C<T> {
    T f;  // Field f has type Object
    void m(Object o) {
        T[] arr = new T[10];
        Class ty = T.class;
        if (o instanceof T) {
            T t = (T)o;
        }
    }
    void mo(C<Integer> x) { }
    void mo(C<String> x) { }
}
```

Illegal in Java

Illegal/doesn't work in Java
Why cannot Java create array T[5]?

- Array assignment requires runtime check:

  ```java
  static void storePerson(Person[] arr, Person x) {
      arr[0] = x;
  }
  ```

  ```java
  Person[] arr = new Student[5];
  storePerson(arr, new Person());
  ```

  Must fail ...

  ```java
  ... else arr[0] now contains a Person, not a Student!
  ```

- For the check, an array must contain its exact element type:

  ```java
  arr
  ```

  | Student
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

But T in C<T> is erased to Object

- That would disable array assignment checks

- Also, no runtime type for constructed types, such as Pair<String,Integer>
- Therefore, Java cannot create array of constructed type:

  ```java
  ... new Pair<String,Integer>[5] ...
  ```

- Java workaround: Use new ArrayList<T>() instead of new T[5]. Why does this work?
Metadata and decompilers

- The `.class` and `.exe` files contain metadata: names and types of fields, methods, classes.
- One can decompile bytecode into programs:
  - For Java, use Jdec or Jad or ...
  - Bad for protecting intellectual property
  - Bytecode obfuscators make decompilation harder

Just-in-time (JIT) compilation

- Bytecode is compiled to real (e.g., x86) machine code at runtime to get speed comparable to C/C++
“Real” machines: The x86

- The x86 is the dominant desktop and laptop (and server?) CPU architecture
- Other architectures (Hitachi H8, ARM, TI) dominate embedded systems, PDAs, mobile phones, … 95-98% of all CPUs
- 8086 → 80286 → 386 → 486 → Pentium → Pentium Pro → Pentium II → Pentium III → Pentium 4 (dying out) → Pentium III → Pentium M → Core → Core 2 (x86-64) → Core 2 Duo → Intel i7

x86 (IA32, i386) registers

![x86 registers diagram]

Figure A.24 IA-32 Register Set. Registers shown are for 32-bit protected-mode operation.
The x86 (IA32, i386) architecture

• A register machine with few registers:
  – esp: stack pointer
  – ebp: base pointer
  – eax, ebx, ecx, edx: int arithmetic registers
  – edi, esi: indexing registers
  – st0-st7: floating-point registers (32-80 bit)
  – mm0-mm7: MMX multimedia registers

• Powerful and very irregular instruction set:
  – Many instructions work only on particular registers
  – Multimedia instructions: SSE, MMX1, ...
  – Instruction length: 1 byte to 15 bytes
  – Mixed-size registers: AH, AL (8 bit); AX (16); EAX (32); RAX (64)
Register machines, principle

- Registers
- Arithmetic and logic unit
- RAM
- Bus
- Disk
- Net
- Display
- etc

Register machines, reality: Pentium M incarnation of x86

- Instruction pipeline:
  - An instruction is decoded in multiple steps
  - Many instr. being decoded simultaneously
- Register renaming:
  - Approx. 40 internal (shadow) registers
  - Dynamically mapped to eax, ebx, ...
- Out-of-order execution:
  - Late instructions may execute before early ones
  - Using different shadow registers
- Multiple integer arithmetic/logic units
A machine code program (Linux x86, tryadd.asm)

• Compute 2+3 and print the result

main:

mov eax, 2 ; put 2 in eax
mov ebx, 3 ; put 3 in ebx
add eax, ebx ; add ebx to eax
push dword eax ; push eax on stack
push dword mystring ; push pointer
call printf ; call C library fcn.
call printf ; call C library fcn.
add esp, byte 8 ; pop 8 bytes from stack
ret

segment DATA

myint dd 1234
mystring db 'The result is ->%d<-', 10, 0
The factorial function (tryfac.asm)

```assembly
fac:
    push    ebp            ; Save ebp on stack
    mov     ebp, esp       ; Save stack pointer in ebp
    mov     eax, [ebp+8]   ; Get argument n
    cmp     eax, 0
    je      .false         ; If n==0 goto .false
    dec     eax            ; else compute n-1 ...
    push    eax            ; and push as argument
    call    fac            ; and compute fac(n-1)
    add     esp, byte 4    ; pop argument
    mul     dword [ebp+8]  ; then multiply eax by n
    jmp     .end

.false: mov     eax, 1         ;
.end:   mov     esp, ebp       ; Restore esp from ebp
        pop     ebp            ; Restore ebp from stack
        ret                    ; Return
```

How good are the JIT compilers?

```csharp
private static int fac(int n) {
    if (n==0)
        return 1;
    else
        return n * fac(n-1);
}
```

CIL bytecode

```csharp
0000: ldarg.0
0001: brtrue 0008
0006: ldc.i4.1
0007: ret
0008: ldarg.0
0009: ldarg.0
000a: ldc.i4.1
000b: sub
000c: call fac(int32)
0011: mul
0012: ret
```

C#
What code from C (gcc 4.2.1)?

```c
int fac(int n) {
    if (n==0)
        return 1;
    else
        return n*fac(n-1);
}
```

```x86-64
_fac:
pushq %rbp
movq %rsp, %rbp
pushq %rbx
subq $8, %rsp
movl %edi, %ebx
movl $1, %eax
testl %edi, %edi
je L4
leal -1(%rbx), %edi
call _fac
imull %ebx, %eax
    ; n*fac(n-1) giving %eax
L4:
addq $8, %rsp
popq %rbx
leave
ret
```

Code from Mono 2.6.1 JIT on MacOS

```asm
00  pushq %rbp
01  movl %esp,%ebp
03  subl $0x08,%esp
06  cmpl $0x00000001,%eax
0a  jne   0x00000013
0c  movl $0x00000001,%eax
11  jmp   0x0000002b
13  movl 0x08(%rbp),%eax
16 subq $0x0c,%rsp
1a  pushq %rax
1b  callq 0x00000000
20  addl $0x10,%esp
23  movl %eax,%ecx
25  movl 0x08(%rbp),%eax
28  imull %ecx,%eax
2b  leave
2c  ret
```

`.NET/CLI bytecode`

```asm
0000: ldarg.0
0001: brtrue 0008
0006: ldc.i4.1
0007: ret
0008: ldarg.0
0009: ldarg.0
000a: ldc.i4.1
000b: sub
000c: call fac(int32)
0011: mul
0012: ret
```

`.NET/CLI bytecode`
How fast are Java and C#?
How predictable are the x86 processors?

- Case studies
  - Matrix multiplication
  - A division intensive series
  - Polynomial evaluation
- Languages
  - C
  - C#
  - Java
- Execution platforms:
  - C: gcc 4.2.1, MacOS
  - C#: Microsoft .NET 4.0 and Mono 2.6
  - Java: Sun Hotspot server (unfortunately not IBM JVM)
- Hardware:
  - Intel Core 2 Duo, 2660 MHz

Naïve matrix multiplication

- Computing $R = A \times B$
  for matrices $A, B, R$

```java
for (r=0; r<rRows; r++) {
  for (c=0; c<rCols; c++) {
    double sum = 0.0;
    for (k=0; k<aCols; k++)
      sum += A.data[r*aCols+k] * B.data[k*bCols+c];
    R.data[r*rCols+c] = sum;
  }
}
```

- Pretty much the same code in Java, C#
- For 80x80, performs $80^3=512,000$ mults
Matrix multiplication benchmark results (us/mult)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C (gcc -03)</td>
<td>702</td>
</tr>
<tr>
<td>C# matmult1 Microsoft</td>
<td>3218</td>
</tr>
<tr>
<td>C# matmult1 Mono</td>
<td>4627</td>
</tr>
<tr>
<td>C# matmult2 Microsoft</td>
<td>1165</td>
</tr>
<tr>
<td>C# matmult2 Mono</td>
<td>1943</td>
</tr>
<tr>
<td>C# matmult3 Microsoft</td>
<td>1575</td>
</tr>
<tr>
<td>C# matmult3 Mono</td>
<td>2888</td>
</tr>
<tr>
<td>Java, Sun Hotspot -server</td>
<td>1180</td>
</tr>
</tbody>
</table>

1.4 ns/iter = 3.7 cycles

C-style unsafe code

array of arrays

Computing a series (division intensive)

• Given M find n such that

\[
\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \geq M
\]

double sum = 0.0;
int n = 0;
while (sum < M) {
    n++;
    sum += 1.0/n;
}

A division takes 20 cpu cycles, dominates all other costs
Division intensive code, benchmark results (ns/iter)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Result (ns/iter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (gcc -O3)</td>
<td>7.9</td>
</tr>
<tr>
<td>C# Microsoft</td>
<td>7.7</td>
</tr>
<tr>
<td>C# Mono</td>
<td>7.6</td>
</tr>
<tr>
<td>Java, Sun Hotspot -server</td>
<td>10.6</td>
</tr>
</tbody>
</table>

20 cycles per division at 2.66 GHz takes 7.5 ns
Division, increment, addition and test done in parallel

Evaluating a polynomial

• Evaluate for given x:

\[ c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \]

```csharp
double res = 0.0;
for (int i=0; i<cs.Length; i++)
    res = cs[i] + x * res;
```
Benchmark results, polynomial of order 1000 (us/eval)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C (gcc -O3)</td>
<td>3.0</td>
</tr>
<tr>
<td>C# Microsoft</td>
<td>3.1</td>
</tr>
<tr>
<td>C# Mono</td>
<td>5.3</td>
</tr>
<tr>
<td>Java, Sun Hotspot -server</td>
<td>3.0</td>
</tr>
</tbody>
</table>

3.0 ns (8 cycles) for
- index increment
- test
- array bounds check
- memory access
- multiplication
- addition

Benchmark results
Normal distribution CDF N(0,1)

```csharp
double expntl = Math.Exp(zabs * zabs * -.5);
double pdf = expntl / root2pi;
if (zabs < cutoff)
    p = expntl * (((((p6 * zabs + p5) * zabs + p4) * zabs + p3) * zabs + p2) * zabs + p1) * zabs + p0) / ((((((q7 * zabs + q6) * zabs + q5) * zabs + q4) * zabs + q3) * zabs + q2) * zabs + q1);
else
    p = pdf / (zabs + 1 / (zabs + 2 / (zabs + 3 / (zabs + 4 / (zabs + .65)))));
```

- Widely used in probability and statistics

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C (gcc -O3)</td>
<td>54</td>
</tr>
<tr>
<td>C# Microsoft</td>
<td>64</td>
</tr>
<tr>
<td>C# Mono</td>
<td>146</td>
</tr>
<tr>
<td>Java, Sun Hotspot -server</td>
<td>69</td>
</tr>
</tbody>
</table>

ns per function call
What's next

• Monday 22 Feb: Scheme, program generation and program specialization
• Friday 26 Feb: Runtime code generation in (Java and) C#