The Java Virtual Machine

Jan Midtgaard
Michael I. Schwartzbach
Aarhus University
Virtual Machines in Compilation

Abstract Syntax Tree

Virtual Machine Code

Native Binary Code

compile

interpret

compile
Virtual Machines in Compilation

Abstract Syntax Tree
  compile

Virtual Machine Code
  interpret  compile

Virtual Machine Code
  interpret  compile

Virtual Machine Code
  interpret  compile

Native Binary Code
Compiling Virtual Machine Code

- Example:
  - gcc translates into RTL, optimizes RTL, and then compiles RTL into native code

- Advantages:
  - facilitates code generators for many targets

- Disadvantage:
  - a code generator must be built for each target
Interpreting Virtual Machine Code

- **Examples:**
  - P-code for Pascal interpreters
  - Postscript code for display devices
  - Java bytecode for the Java Virtual Machine

- **Advantages:**
  - easy to generate code
  - the code is architecture independent
  - bytecode can be more compact

- **Disadvantage:**
  - poor performance (naively 5-100 times slower)
The instruction set may be more or less high-level.

A balance must be found between:
- the work of the compiler
- the work of the interpreter

In the extreme case, there is only one instruction:
- compiler guy: `execute "program"`
- interpreter guy: `print "result"`

The resulting sweet spot involves:
- doing as much as possible at compile time
- exposing the program structure to the interpreter
- minimizing the size of the generated code
- being able to verify security & safety policies on compiled code
Components of the JVM:

- stack (per thread)
- heap
- constant pool
- code segment
- program counter (per thread)

(we ignore multiple threads in this presentation)
The Java Stack

- The stack consists of frames:

- The number of local slots in and the size of a frame are fixed at compile-time.
- Note how a frame of the call stack contains smaller operand stack for storing temporary values.

Diagram:
- Stack consists of frames:
  - Operand stack
  - Locals
  - Arguments
  - This

Diagram: Stack with sp and lsp pointers.
The Java Heap

- The *heap* consists of *objects*:
The Java Constant Pool

- The *constant pool* consists of all *constant data*:
  - numbers
  - strings
  - symbolic names of classes, interfaces, and fields
The Java Code Segment

- The *code segment* consists of *bytecodes* of variable sizes:
Java Bytecodes

- A **bytecode** instruction consists of:
  - a one-byte opcode
  - a variable number of arguments (offsets or pointers to the constant pool)
- It consumes and produces some stack elements
- Constants, locals, and stack elements are typed:
  - addresses (a)
  - primitive types (i,c,b,s,f,d,l)
Class Files

- Java compilers generate class files:
  - magic number (0xCAFEBABE)
  - minor version/major version
  - constant pool
  - access flags
  - this class
  - super class
  - interfaces
  - fields
  - methods
  - attributes (extra hints for the JVM or other applications)
Class Loading

- Classes are loaded lazily when first accessed
- Class name must match file name
- Super classes are loaded first (transitively)
- The bytecode is verified
- Static fields are allocated and given default values
- Static initializers are executed
From Methods to Bytecode

- A simple Java method:

```java
public int Abs(int i) {
    if (i < 0)
        return(i * -1);
    else
        return(i);
}
```

```
.method public Abs(I)I // int argument, int result
.limit stack 2 // stack with 2 locations
.limit locals 2 // space for 2 locals
    // --locals-- --stack---
    iload_1 // [ x -3 ] [ -3 * ]
    ifge Label1 // [ x -3 ] [ * * ]
    iload_1 // [ x -3 ] [ -3 * ]
    iconst_m1 // [ x -3 ] [ -3 -1 ]
    imul // [ x -3 ] [ 3 * ]
    ireturn // [ x -3 ] [ * * ]
Label1:
    iload_1
    ireturn
.end method
```

- Comments show trace of: x.Abs(-3)
A Sketch of a Bytecode Interpreter

The core of a VM consists of a fetch-decode-execute loop:

```java
pc = code.start;
while(true)
    {npc = pc + instruction_length(code[pc]);
     switch (opcode(code[pc]))
     { case ILOAD_1: push(locals[1]);
             break;
     case ILOAD: push(locals[code[pc+1]]);
                break;
     case ISTORE: t = pop();
                 locals[code[pc+1]] = t;
                 break;
     case IADD: t1 = pop(); t2 = pop();
                 push(t1 + t2);
                 break;
     case IFEQ: t = pop();
                 if (t==0) npc = code[pc+1];
                 break;
     ...
     }
    pc = npc;
    }
```
The JVM has 256 instructions for:

- arithmetic operations
- branch operations
- constant loading operations
- locals operations
- stack operations
- class operations
- method operations

See the JVM specification for the full list
## Arithmetic Operations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ineg</td>
<td>$[...:i] \rightarrow [...:-i]$</td>
<td></td>
</tr>
<tr>
<td>i2c</td>
<td>$[...:i] \rightarrow [...:i \mod 65536]$</td>
<td></td>
</tr>
<tr>
<td>iadd</td>
<td>$[...:i:j] \rightarrow [...:i+j]$</td>
<td></td>
</tr>
<tr>
<td>isub</td>
<td>$[...:i:j] \rightarrow [...:i-j]$</td>
<td></td>
</tr>
<tr>
<td>imul</td>
<td>$[...:i:j] \rightarrow [...:i*j]$</td>
<td></td>
</tr>
<tr>
<td>idiv</td>
<td>$[...:i:j] \rightarrow [...:i/j]$</td>
<td></td>
</tr>
<tr>
<td>irem</td>
<td>$[...:i:j] \rightarrow [...:i%j]$</td>
<td></td>
</tr>
<tr>
<td>iinc k i</td>
<td>$[...] \rightarrow [...]$</td>
<td>locals[k] = locals[k] + i</td>
</tr>
</tbody>
</table>
### Branch Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>goto L</code></td>
<td>Jump to label <code>L</code> without further condition</td>
</tr>
<tr>
<td><code>ifeq L</code></td>
<td>Branch if the condition <code>i</code> is equal to zero</td>
</tr>
<tr>
<td><code>ifne L</code></td>
<td>Branch if the condition <code>i</code> is not equal to zero</td>
</tr>
<tr>
<td><code>ifnull L</code></td>
<td>Branch if the condition <code>a</code> is null</td>
</tr>
<tr>
<td><code>ifnonnull L</code></td>
<td>Branch if the condition <code>a</code> is not null</td>
</tr>
</tbody>
</table>
Branch Operations

- if_icmpeq L  [...:i:j] → [...]
  branch if i==j

- if_icmpne L

- if_icmpge L

- if_icmplt L

- if_acmpeq L  [...:a:b] → [...]
  branch if a==b

- if_acpmne L
Constant Loading Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iconst_0</td>
<td>[... → [...:0]</td>
</tr>
<tr>
<td>iconst_1</td>
<td>[... → [...:1]</td>
</tr>
<tr>
<td>iconst_5</td>
<td>[... → [...:5]</td>
</tr>
<tr>
<td>aconst_null</td>
<td>[... → [...:null]</td>
</tr>
<tr>
<td>ldc i</td>
<td>[... → [...:i]</td>
</tr>
</tbody>
</table>

More precisely, the argument of ldc is an index into the constant pool of the current class, and the constant at that index is pushed.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldc s</td>
<td>[... → [...:String(s)]</td>
</tr>
</tbody>
</table>

Again, the argument to ldc is actually an index into the constant pool.
## Locals Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload k</td>
<td></td>
<td>[...] → [\ldots:locals[k]]</td>
</tr>
<tr>
<td>istore k</td>
<td></td>
<td>[\ldots:i] → [\ldots] (\text{locals}[k]=i)</td>
</tr>
<tr>
<td>aload k</td>
<td></td>
<td>[\ldots] → [\ldots:locals[k]]</td>
</tr>
<tr>
<td>astore k</td>
<td></td>
<td>[\ldots:a] → [\ldots] (\text{locals}[k]=a)</td>
</tr>
</tbody>
</table>
Field Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>getfield f sig</td>
<td>[...:a] → [...:a.f]</td>
<td></td>
</tr>
<tr>
<td>putfield f sig</td>
<td>[...:a:v] → [...]</td>
<td>a.f=v</td>
</tr>
<tr>
<td>getstatic f sig</td>
<td>[...] → [...:C.f]</td>
<td></td>
</tr>
<tr>
<td>putstatic f sig</td>
<td>[...:v] → [...]</td>
<td>C.f=v</td>
</tr>
</tbody>
</table>

More precisely, the argument to these operations is an index in the constant pool which must contain the signature of the corresponding field.
# Stack Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dup</td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td></td>
</tr>
<tr>
<td>swap</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>dup_x1</td>
<td></td>
</tr>
<tr>
<td>dup_x2</td>
<td></td>
</tr>
</tbody>
</table>

- `dup`: `[...:v] → [...:v:v]`
- `pop`: `[...:v] → [...]`
- `swap`: `[...v:w] → [...:w:v]`
- `nop`: `[...,] → [...]`
- `dup_x1`: `[...:v:w] → [...:w:v:w]`
- `dup_x2`: `[...:u:v:w] → [...:w:u:v:w]`
new C         
[...:] → [...:a] 

instanceof C
[...:a] → [...:i] 
if (a==null) i==0 
else i==(type(a)≤C) 

castcast C     
[...:a] → [...:a] 
if (a!=null) && !type(a)≤C) 
throw ClassCastException
**Method Operations**

```java
invokevirtual name sig
    [...:a:v₁:.:.vₙ] → [...(:v)]
m=lookup(type(a),name,sig)
push frame of size m.locals+m.stacksize
locals[0]=a
locals[1]=v₁
...
locals[n]=vₙ
pc=m.code
```

- `invokestatic`
- `invokespecial`
- `invokeinterface`
## Method Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack Pre</th>
<th>Stack Post</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ireturn</code></td>
<td><code>[...:i]</code></td>
<td><code>[...]</code></td>
<td>return i and pop stack frame</td>
</tr>
<tr>
<td><code>areturn</code></td>
<td><code>[...:a]</code></td>
<td><code>[...]</code></td>
<td>return a and pop stack frame</td>
</tr>
<tr>
<td><code>return</code></td>
<td><code>[...]</code></td>
<td><code>[...]</code></td>
<td>pop stack frame</td>
</tr>
</tbody>
</table>
A Java Method

public boolean member(Object item)
{
    if (first.equals(item))
        return true;
    else if (rest == null)
        return false;
    else
        return rest.member(item);
}
Generated Bytecode

```
.method public member(Ljava/lang/Object;)Z
.limit locals 2 // locals[0] = x
    // locals[1] = item
.limit stack 2 // initial stack [ * * ]
aload_0 // [ x * ]
getfield Cons/first Ljava/lang/Object;
    // [ x.first *]
aload_1 // [ x.first item]
invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
    // [bool *]
ifeq else_1 // [ * * ]
iconst_1 // [ 1 * ]
ireturn // [ * * ]
else_1:
aload_0 // [ x * ]
getfield Cons/rest LCons; // [ x.rest * ]
aconst_null // [ x.rest null]
if_acmpne else_2 // [ * * ]
iconst_0 // [ 0 * ]
ireturn // [ * * ]
else_2:
aload_0 // [ x * ]
getfield Cons/rest LCons; // [ x.rest * ]
aload_1 // [ x.rest item ]
invokevirtual Cons/member(Ljava/lang/Object;)Z
    // [ bool * ]
ireturn // [ * * ]
.end method
```
Bytecode Verification

- Bytecode cannot be trusted to be well-behaved
- Before execution, it must be verified
- Verification is performed:
  - at class loading time
  - at runtime
- A Java compiler must generate verifiable code
The first 4 bytes of a class file must contain the magic number \texttt{0xCAFEBAFB}

The bytecodes must be syntactically correct
Verification: Constants and Headers

- Final classes are not subclassed
- Final methods are not overridden
- Every class except `Object` has a superclass
- All constants are legal
- Field and method references have valid signatures
Verification: Instructions

- Branch targets are within the code segment
- Only legal offsets are referenced
- Constants have appropriate types
- All instructions are complete
- Execution cannot fall of the end of the code
Verification: Dataflow Analysis and Type Checking

- At each program point, the stack always has the same size and types of objects
- No uninitialized locals are referenced
- Methods are invoked with appropriate arguments
- Fields are assigned appropriate values
- All instructions have appropriate types of arguments on the stack and in the locals
**Verification: Gotcha**

```
.method public static main([Ljava/lang/String;)V
  .throws java/lang/Exception
  .limit stack 2
  .limit locals 1
  ldc -21248564
  invokevirtual java/io/InputStream/read()I
  return
```

java Fake

Exception in thread "main" java.lang.VerifyError:
  (class: Fake, method: main signature: ([Ljava/lang/String;)V)
Expecting to find object/array on stack
Verification: Gotcha Again

```
.method public static main([Ljava/lang/String;)V
    .throws java/lang/Exception
    .limit stack 2
    .limit locals 2
    iload_1
    return
```

```
java Fake

Exception in thread "main" java.lang.VerifyError:  
(class: Fake, method: main signature: ([Ljava/lang/String;)V)  
Accessing value from uninitialized register 1
```
ifeq A
dc 42
goto B
A:
dc "fortytwo"
B:

dc Fake

Exception in thread "main" java.lang.VerifyError:
(class: Fake, method: main signature: ([Ljava/lang/String;)V
Mismatched stack types
Verification: Gonna Getcha Every Time

A:
iconst_5
goto A

java Fake

Exception in thread "main" java.lang.VerifyError:
(class: Fake, method: main signature: ([Ljava/lang/String;)V
Inconsistent stack height 1 != 0
Alternative: Proof-Carrying Code

- Elegant verification approach to enforce safety and security policies
  - based on theorem proving methods
- E.g., allows distribution of native code while maintaining the safety guarantees of VM code
- No trust in the originator is needed
JVM Implementation

- A naive bytecode interpreter is slow
- State-of-the-art JVM implementations are not:

![C++ vs. Java Performance Chart](http://kano.net/javabench)
### JVM Implementation

- A naive bytecode interpreter is slow
- State-of-the-art JVM implementations are not:

#### C++ vs. Java Performance

<table>
<thead>
<tr>
<th>Task</th>
<th>GCC (Intel 386+)</th>
<th>GCC (Intel 686+)</th>
<th>Server JVM</th>
<th>Client JVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ackermann</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fibonacci</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method call</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object creation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random gen.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sieve</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String concatenation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sumcol</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word count</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

However: take micro-benchmarks with a grain of salt

[http://kano.net/javabench](http://kano.net/javabench)
Just-In-Time Compilation

- Exemplified by SUN’s HotSpot JVM

- Bytecode fragments are compiled at runtime
  - targeted at the native platform
  - based on runtime profiling
  - customization is possible

- Offers more opportunities than a static compiler

- It needs to be fast as it happens at run-time
Other Java Bytecode Tools

- assembler (*jasmin*)
- disassembler (*javap*)
- decompiler (*cavaj*, *mocha*, *jad*)
- obfuscator (*dozens of these...*)
- analyzer (*soot*)