Compilation 2008

The Java Virtual Machine

Klaus Ostermann
Michael I. Schwartzbach
University of Aarhus
Virtual Machines in Compilation

- Abstract Syntax Tree
  - compile

- Virtual Machine Code
  - interpret, compile

- Native Binary Code
Virtual Machines in Compilation

Abstract Syntax Tree
   \[\text{compile}\]
   \[\text{Virtual Machine Code}\]
   \[\text{interpret} \quad \text{compile}\]
   \[\text{Virtual Machine Code}\]
   \[\text{interpret} \quad \text{compile}\]
   \[\text{Virtual Machine Code}\]
   \[\text{interpret} \quad \text{compile}\]
   \[\text{Native Binary Code}\]
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)
Designing A Virtual Machine

- 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.
The Java Heap

- The *heap* consists of *objects*: 

- **runtime type:**
- **fields:**
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 (0xCAFEBAFB)
  - 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
Class Loaders

- A class loader is an object that is responsible for loading classes
- Each class loader is an instance of the abstract class java.lang.ClassLoader
- Every Class object contains a reference to the ClassLoader that defined it
- Each class loader has a parent class loader
  - First try parent class loader if class is requested
  - There is a bootstrap class loader which is the root of the classloader hierarchy
- Class loaders are a quite powerful extension mechanism in Java
  - Loading classes from other sources
  - Transforming classes during loading
  - java.lang.reflect.Proxy
class NetworkClassLoader extends ClassLoader {
    String host;
    int port;

    public Class findClass(String name) {
        byte[] b = loadClassData(name);
        return defineClass(name, b, 0, b.length);
    }

    private byte[] loadClassData(String name) {
        // load the class data from the connection
        ...
    }
}

ClassLoader loader = new NetworkClassLoader(host, port);
Object main = loader.loadClass("Main", true).newInstance();
...
A simple Java method:

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

Comments show trace of: `x.Abs(-3)`
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>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ineg</code></td>
<td><code>[...:i] → [...:-i]</code></td>
</tr>
<tr>
<td><code>i2c</code></td>
<td><code>[...:i] → [...:i%65536]</code></td>
</tr>
<tr>
<td><code>iadd</code></td>
<td><code>[...:i:j] → [...:i+j]</code></td>
</tr>
<tr>
<td><code>isub</code></td>
<td><code>[...:i:j] → [...:i-j]</code></td>
</tr>
<tr>
<td><code>imul</code></td>
<td><code>[...:i:j] → [...:i*j]</code></td>
</tr>
<tr>
<td><code>idiv</code></td>
<td><code>[...:i:j] → [...:i/j]</code></td>
</tr>
<tr>
<td><code>irem</code></td>
<td><code>[...:i:j] → [...:i%j]</code></td>
</tr>
<tr>
<td><code>iinc k i</code></td>
<td><code>[...] → [...]</code></td>
</tr>
<tr>
<td></td>
<td><code>locals[k]=locals[k]+i</code></td>
</tr>
</tbody>
</table>
# Branch Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto L</td>
<td>([\ldots] \rightarrow [\ldots])</td>
</tr>
<tr>
<td></td>
<td>branch always</td>
</tr>
<tr>
<td>ifeq L</td>
<td>([\ldots:i] \rightarrow [\ldots])</td>
</tr>
<tr>
<td></td>
<td>branch if (i == 0)</td>
</tr>
<tr>
<td>ifne L</td>
<td>([\ldots:i] \rightarrow [\ldots])</td>
</tr>
<tr>
<td></td>
<td>branch if (i != 0)</td>
</tr>
<tr>
<td>inull L</td>
<td>([\ldots:a] \rightarrow [\ldots])</td>
</tr>
<tr>
<td></td>
<td>branch if (a == \text{null})</td>
</tr>
<tr>
<td>ifnonnull L</td>
<td>([\ldots:a] \rightarrow [\ldots])</td>
</tr>
<tr>
<td></td>
<td>branch if (a != \text{null})</td>
</tr>
</tbody>
</table>
Branch Operations

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

if_icmpne L

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

if_acmpne L
Constant Loading Operations

- `icnst_0`  
  `[[...]] \rightarrow [[...:0]]`

- `icnst_1`  
  `[[...]] \rightarrow [[...:1]]`

- `icnst_5`  
  `[[...]] \rightarrow [[...:5]]`

- `aconst_null`  
  `[[...]] \rightarrow [[...:null]]`

- `ldc i`  
  `[[...]] \rightarrow [[...:i]]`

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.

If index points to a string constant:

- `ldc s`  
  `[[...]] \rightarrow [[...:adressof(s)]]`
## Locals Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload k</td>
<td>Load integer value into local variable k</td>
<td>[... ] → [... : locals[k]]</td>
</tr>
<tr>
<td>istore k</td>
<td>Store integer value i into local variable k</td>
<td>[... :i ] → [...]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>locals[k]=i</td>
</tr>
<tr>
<td>aload k</td>
<td>Load object reference into local variable k</td>
<td>[... ] → [... : locals[k]]</td>
</tr>
<tr>
<td>astore k</td>
<td>Store object reference a into local variable k</td>
<td>[... :a ] → [...]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>locals[k]=a</td>
</tr>
</tbody>
</table>
Field Operations

getfield f sig [...:a] → [...:a.f]

putfield f sig [...:a:v] → [...]
  a.f=v

getstatic f sig [...] → [...:C.f]

putstatic f sig [...:v] → [...]
  C.f=v

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>Stack before</th>
<th>Stack after</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dup</code></td>
<td><code>[...:v]</code></td>
<td><code>[...:v:v]</code></td>
</tr>
<tr>
<td><code>pop</code></td>
<td><code>[...:v]</code></td>
<td><code>[...]</code></td>
</tr>
<tr>
<td><code>swap</code></td>
<td><code>[...v:w]</code></td>
<td><code>[...:w:v]</code></td>
</tr>
<tr>
<td><code>nop</code></td>
<td><code>[...]</code></td>
<td><code>[...]</code></td>
</tr>
<tr>
<td><code>dup_x1</code></td>
<td><code>[...:v:w]</code></td>
<td><code>[...:w:v:w]</code></td>
</tr>
<tr>
<td><code>dup_x2</code></td>
<td><code>[...:u:v:w]</code></td>
<td><code>[...:w:u:v:w]</code></td>
</tr>
</tbody>
</table>
Class Operations

new C  
[...]:a

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

checkcast C  
[...]:a  →  [...]:a
if (a!=null)  !type(a)<=C
throw ClassCastException
Java Virtual Machine

Method Operations

invokevirtual name sig

\[\ldots : a : v_1 : \ldots : v_n \rightarrow \ldots (: v)\]

m = lookup(type(a), name, sig)

push frame of size m.locals + m.stacksize

locals[0] = a

locals[1] = v_1

\ldots

locals[n] = v_n

pc = m.code

 invokestatic

 invokespecial

 invokeinterface
## Method Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack Before</th>
<th>Stack After</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ireturn</code></td>
<td><code>[...:i]</code></td>
<td><code>[...]</code></td>
<td>Return <code>i</code> and pop stack frame</td>
</tr>
<tr>
<td><code>areturn</code></td>
<td><code>[...:a]</code></td>
<td><code>[...]</code></td>
<td>Return <code>a</code> 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>
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 * ]
aconst_1          // [ x.rest item ]
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
Verification: Syntax

- The first 4 bytes of a class file must contain the magic number `0xCAFEBAEBE`
- 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 off 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
Java Virtual Machine

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

```java
.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
```
Verification: Gotcha Once More

```
ifeq A
ldc 42
goto B
A:
ldc "fortytwo"
B:
```

```
java 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: Code Signing

- Code is signed with cryptographic methods by the vendor
- Code receiver verifies signature
- User must trust vendor
- Code may still violate safety policy
  - Bugs in the code
  - Malicious programmers at the vendor
  - …
Alternative: Proof-Carrying Code

- Very elegant verification approach to enforce safety & 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
A naive bytecode interpreter is slow
State-of-the-art JVM implementations are not:

![C++ vs. Java Performance Chart](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
Other Java Bytecode Tools

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