Virtual Machine Design
Lecture 3: Interpretation and Execution

Antero Taivalsaari
September 2003
At the heart of a virtual machine is the **intermediate code interpreter**. This lecture will explain the key ideas and algorithms for interpreter design and implementation.
Essential Components of a (Java) Virtual Machine

- Class Loader and JAR Reader
- Interpreter and Execution Stacks
- Threading System and Thread Scheduler
- Native Interface
- Internal Runtime Structures
- Memory System and Garbage Collector
- Verifier
- Compiler (optional)
Interpretation and Execution
Background

• Executing source code directly would be too expensive and difficult.
  – Parsing of source code takes a lot of time and space.
  – In general, source code is intended to be human-readable; it is not intended for direct execution.

• Most virtual machines use some kind of an intermediate representation to store programs.

• Most virtual machines use an interpreter to execute code that is stored in the intermediate representation.
Two Kinds of Interpreters

- Virtual machines commonly use two kinds of interpreters:

① Command-line interpreter ("outer" interpreter)
- Reads and parses instructions in source code form (textual representation).
- Only needed in those systems that can read in source code at runtime.

② Instruction interpreter ("inner" interpreter)
- Reads and executes instructions in some intermediate execution format such as bytecodes.
Two Kinds of Interpreters

Source code → Outer interpreter → Intermediate code → Inner interpreter
Comments on Outer Interpreters

• The complexity of an outer interpreter depends on the programming language.
  – For high-level languages, a fairly advanced compiler (lexical analyzer/parser/code generator) is needed.
  – For syntactically simpler languages such as Lisp or Forth, the outer interpreter can be very simple.

  - The “read-eval-print” loop of Lisp:
    (print (eval (read)))
  - Forth's outer interpreter:
    BEGIN query interpret AGAIN

• Non-interactive programming languages do not require any outer interpreter at all.
  – For instance, a Java virtual machine only needs to read in class files and JAR files in binary form.
Comments on Inner Interpreters

• The heart of the virtual machine is the inner interpreter.

• The behavior of the inner interpreter:
  1) reads the current instruction,
  2) increments the instruction pointer,
  3) parses and executes the instruction,
  4) goes back to (1) to read the next instruction
A Minimal Inner Interpreter

• A simple interpreter written in C:

```c
int* ip; /* instruction pointer */
while (true) {
    ((void (*)(()))*ip++)();
}
```
Components of an Interpreter

- Interpreters usually have the following components:

  **Interpreter loop**

  ```c
  while (true) {
      ((void (*)())*ip++)();
  }
  ```

  **Instruction set**

  add, mul, sub, ...
  load, store, branch, ...
  ...

  **Virtual registers**

  ```
  ip fp ...
  sp lp
  ```

  **Execution stacks**

  ```
  Operand stack
  Execution stack
  ...
  ```
Virtual Registers

- *Virtual registers* hold the state of the interpreter during execution. Typical virtual registers:
  - *ip*: instruction pointer
    - Points to the current (or next) instruction to be executed.
  - *sp*: stack pointer
    - Points to the topmost item in the operand stack.
  - *fp*: frame pointer
    - Points to the topmost frame (activation record) in the execution stack (call stack).
  - *lp*: local variable pointer
    - Points to the beginning of the local variables in the execution stack.
  - *up*: current thread pointer
Execution Stacks

• In order to support method (subroutine) calls and proper control flow, an execution stack is typically needed.  
  – Also known as the call stack.

• Execution stack holds the stack frames (activation records) at runtime.  
  – Allows the interpreter to invoke methods/subroutines and to return to correct locations once a method call ends.  
  – Each thread in the VM usually needs its own execution stack.

• Some VMs use a separate operand stack to store parameters and operands.
Fundamental Interpretation Techniques
Interpretation: Fundamental Techniques

- **Token-based interpretation**
  - P Code (UCSD Pascal)
  - Bytecode interpretation (Smalltalk, Java)

- **Address-based interpretation**
  - Indirect threaded code (Forth)
  - Direct threaded code (Forth)
  - Subroutine threading

- **String/pattern-based interpretation**
  (PostScript, Snobol, ...)

Taxonomy of Interpreters

Interpreters
- Token-based interpreters
  - Bytecode interpreters
- Address-based interpreters
  - Direct threaded code interpreters
  - Indirect threaded code interpreters
- Pattern-based interpreters
  - Subroutine threaded code interpreters
Token-Based Interpretation

• In token-based interpreters, the fundamental instruction unit is a *token*.
  – Token is a predefined numeric value that represents a certain instruction.
    • E.g., 1 = LOAD LITERAL, 2 = ADD, 3 = MULTIPLY, ...
  – Token values are independent of the underlying hardware or operating system.

• The most common subcase:
  – In a *bytecode interpreter*, instruction (token) width is limited to 8 bits.
  – Total instruction set limited to 256 instructions.
  – Bytecode interpreters are very commonly used, e.g., for Smalltalk, Java, and many other interpreted programming languages.
## Token-Based Code: Examples

Some Java bytecodes:

- `iload_1`
- `iconst_1`
- `iadd`
- `ishl`
- `istore_1`
- `return`

<table>
<thead>
<tr>
<th>Control flow</th>
<th>Token 1</th>
<th>Token 2</th>
<th>Token 3</th>
<th>...</th>
<th>...</th>
<th>Token n</th>
</tr>
</thead>
</table>

Code is represented as linear lists that contain fixed-size tokens. In bytecode, token width is 8 bits.
A Simple Bytecode Interpreter

```java
void Interpreter() {
    while (true) {
        byte token = (byte)*ip++;

        switch (token) {
            case INSTRUCTION_1:
                break;
            case INSTRUCTION_2:
                break;
            case INSTRUCTION_3:
                break;
        }
    }
}
```
Address-based Interpreters

• In address-based interpreters, the fundamental instruction unit is an address rather than a token.

• Address-based interpreters can be more efficient than token-based, especially when written in machine code.
  – No token decoding overhead.
  – Can jump directly from one instruction to the next.

• The term *threaded code* refers to a code representation where every instruction is implicitly a function call to another address.
Threaded Code: The Basic Idea

In threaded code, code is stored as a linear list of addresses.

Every instruction (word) contains an address that represents a function to execute.

Some words are *primitives*; primitives are executed directly by calling the address stored in the word itself.

Other words are *non-primitives*; when executing a non-primitive, control flow is transferred to another list of addresses.

Control flow

16 or 32 bits wide
Threaded Code: Example

Equivalent Forth code:
: foo 100 200 + ;

Explanation:

- Read the literal value 100 from the instruction stream and push it onto the operand stack.
- Read the literal value 200 from the instruction stream and push it onto the operand stack.
- Add two topmost items in the operand stack.
- Return control back to caller, leaving the result on the operand stack.
A Simple Threaded Code Interpreter

void Interpreter() {
    while (true) {
        word* address = (word*)*ip++;
        if (isPrimitiveInstruction(address)) {
            /* Run primitive */
            ((void (*)())*address)();
        } else {
            /* Execute subroutine */
            pushReturn(ip);
            ip = address;
        }
    }
}
Direct vs. Indirect Threaded Code

- There are three commonly used forms of threaded code:
  - Direct threaded code
  - Indirect threaded code
  - Subroutine threaded code

- Direct threaded code is most efficient, but it cannot be implemented in a portable fashion using C or other high-level languages.
Direct Threaded Code

• In direct threaded code, there is no centralized interpreter loop at all.

• When the execution of a primitive ends, control is transferred directly to the next instruction by performing a “computed goto” instruction.

```c
void doSomething() {
    ... do something ...
    /* Transfer control to the next instruction */
goto (*ip++);
}
```

(instruction pointer has been pre-incremented to point here.)
Indirect Threaded Code

• In indirect threaded code, there is an extra level of indirection in the code representation.
  – Primitive functions are not called directly.
  – Instead, there is a common function for calling all primitives.

• Indirect threaded code is easier to implement in a portable fashion and has other benefits.
  – Commonly used, e.g., in Forth interpreters.
Subroutine Threaded Code

• There is a third, less commonly used variant of threaded code.
• In “subroutine threaded code”, the machine-specific subroutine call and return instructions are used for calling the primitives:

```
...  
JSR PushLiteral()  
100
JSR PushLiteral()  
200
JSR Add()  
RET
```
String-based Interpreters

• Some interpreters execute code by parsing strings or by performing pattern matching.
  – PostScript, SNOBOL, ...
  – Command line interpreters (csh, bash, ...)

• String-based interpreters are usually too slow for virtual machine implementation.
  – Needed mainly for implementing specialized languages.
  – We ignore them in this presentation...
Instruction Sets

• Each virtual machine typically has its own instruction set.

• These instruction sets are fairly similar to instruction sets of hardware CPUs.

• The types of instructions are generally the same as in CPUs:
  • Local variable load and store operations
  • Constant value load operations
  • Array load and store operations
  • Arithmetic operations (add, sub, mul, div, ...)
  • Logical operations (and, or, xor, ...)
  • Type conversions
  • Conditional and unconditional branches
  • Method invocations and returns
  • ...
Stack-Oriented vs. Register-Oriented Instruction Sets

• Two types of instruction sets:
  1. In *stack-oriented* instruction sets, operands to most instructions are passed in an operand stack; this stack can grow and shrink dynamically as needed.
  2. In *register-oriented* instruction sets, operands are accessed via “register windows”: fixed-size areas that are allocated automatically upon method calls.

• Most virtual machines (unlike modern CPUs!) use a stack-oriented instruction set.
  – Stack machines are generally simpler to implement.
  – No problems with “running out of registers”; the instruction set can be smaller.
  – Less encoding/decoding needed to parse register numbers.
Example: The Java Bytecode Interpreter

- The JVM uses a straightforward stack-oriented bytecode instruction set with 200 instructions.
  - Fairly similar to the Smalltalk bytecode set, except that in Java primitive data types are not objects.

- One execution stack is required per each Java thread.
  - No separate operand stack; operands are kept on top of the current stack frame.

- Four virtual registers are commonly used:
  - \( ip \) (instruction pointer): points to current instruction
  - \( sp \) (stack pointer): points to the top of the stack
  - \( fp \) (frame pointer): provides fast access to stack frame
  - \( lp \) (locals pointer): provides fast access to local variables
Example: The Java Virtual Machine Instruction Set

(Note: "_quick" bytecodes are non-standard and implementation-dependent)
JVM Instruction Formats

1. Most Java bytecodes do not require any parameters from the instruction stream.
   - They operate on the values provided on the execution stack (e.g., IADD, IMUL, ...)

2. Some bytecodes read an additional 8-bit parameter from the instruction stream.
   - For instance, NEWARRAY, LDC, *LOAD, *STORE

3. Many bytecodes read additional 16 bits from the instruction stream.
   - INVOKE* instructions, GET/PUTFIELD, GET/PUTSTATIC, branch instructions, ...

4. Three instructions are varying-length.
   - LOOKUPSWITCH, TABLESWITCH, WIDE
Example: BIPUSH

BIPUSH     8-bit parameter

Operand stack: ... => value, ...

- BIPUSH: Push a byte value onto the stack.
- The instruction reads an 8-bit parameter value from the instruction stream. The 8-bit value is sign-extended to a 32-bit value, and that value is pushed onto the operand stack.
Example: IFNULL

<table>
<thead>
<tr>
<th>IFNULL</th>
<th>16-bit parameter</th>
</tr>
</thead>
</table>

Operand stack: ..., value => ...

- IFNULL: Branch if reference is null.
- The instruction pops value off the operand stack, and checks if the value is NULL.
- The 16-bit parameter contains a branch offset that is added to the instruction pointer if value is NULL.
- Otherwise, execution continues normally from the next instruction.
Example: GETFIELD

GETFIELD 16-bit parameter

Operand stack: ..., objectref => ..., value

- GETFIELD: Get the value of an instance variable.
- The 16-bit parameter points to a constant pool entry that determines the field (and class of the field) that is to be accessed.
- The field's slot index is added to the objectref parameter (popped from the operand stack) to form the physical address of the field.
- Field value is pushed onto the operand stack.
Another Instruction Set Example: Smalltalk-80

- See separate document...
Another Instruction Set: Self

• The programming language Self has a radically different bytecode set.

• Only 8 basic bytecodes!
  – SELF, LITERAL, NON-LOCAL RETURN, DIRECTEE, SEND, IMPLICIT SELF SEND, RESEND, INDEX-EXTENSION
  – 3 bits of the 8-bit bytecode reserved for the opcode itself.
  – 5 bits of the 8-bit bytecode token are reserved for parameter info.

• The rest of the primitives are implemented as native functions that are invoked using the normal message sending bytecode SEND.
Implementing Interpreter Instructions
Accessing “Inline” Parameters

• In most instruction sets, there are instructions that need to read “inlined” parameters from the instruction stream.

• For instance, branch instructions typically need to read the branch offset:

  ```
  IFNULL 0x0010
  ```

• Remember the possible endianness problems!
  – The individual bytes of 16/32/64-bit numbers are represented in different order in different CPUs!
  – In Java, all numbers in class files are *big-endian*; if a machine-specific endianness was used, Java class files wouldn't be portable across different machines.
Accessing “Inline” Parameters

- Inline parameters are generally very easy to access.
- Use the instruction pointer to determine the location.

```c
void UnconditionalBranch() {
    int branchOffset = *ip;
    ip += branchOffset;
}
```

(instruction pointer has been pre-incremented to point here.)
Stack Frames, Activation Records and Exception Handling
Stack Frames (Activation Records) in Java

- Each time you do a method call (execute an INVOKE* bytecode), a new stack frame needs to be created.
  - The stack frame contains the method call arguments, has space for the local variables, and contains some administrative data as well.

- When the method returns, the stack frame is popped (removed) from the execution stack.
  - The values of the virtual registers (ip, sp, fp, ...) are restored so that the earlier method can continue to run.

- Each Java thread has its own execution stack.

- There is no separate operand stack in a JVM.
  - Operands of bytecodes are kept on top of the current stack frame.
Example: Stack Frames in KVM 1.0

Stack growth direction

- SP (top of Java stack)
- Operand stack starts here
- Pointer to local variables inside the frame
- Pointer to locked object (if synchronized call)
- Previous instruction pointer
- Pointer to previous stack frame
- What to do upon returning from method
- Pointer to the current constant pool
- FP (frame pointer)
- LP (parameters + local variables)
Chunky Stacks

• Execution stacks can consume a lot of space.
• Unlike many VMs in which execution stacks are of fixed size, KVM has “chunky stacks”.
  – Space for stacks is allocated dynamically from the heap.
  – Old, unused stack chunks are automatically reclaimed by the garbage collector.

• Default stack chunk size is small (½ kilobyte).
  – Because of chunky stacks, KVM can support an exceptionally large number of Java threads (100 threads in less than a 100 kilobyte heap).

• Ideal for embedded systems.
  – Fixed size stacks would waste a lot of memory!
Activation Records in Other VMs

- Not all programming languages use or need stack-allocated activation records!
- In Smalltalk, method block instances are allocated dynamically from the heap.
  - Every method invocation creates garbage!
  - The actual VM implementation may optimize the semantics and use stack allocation instead.
- In Forth, there are no activation records.
  - The only thing that needs to be pushed onto the execution stack (called the 'return stack' in Forth) is the current instruction pointer.
  - Parameters and return values are stored explicitly in a separate operand stack.
Exception Handling Support

• The Java programming language supports exception handling:

```java
try {
    ... do something ...
} catch (IOException e1) {
    ... handle I/O failure ...
} catch (MyException e2) {
    ... handle app-specific failure ...
}
```

• When an exception occurs, control flow must return to the exception handler no matter how many levels of nested subroutine/method calls have been made inside the `try` block!!
Implementing Exception Handling

• When an exception occurs:
  ① The virtual machine scans the execution stack for matching *exception handlers*.
    • Scanning is performed by comparing the exception handler type and the bytecode range in which the handler is valid.
    • Scanning is started from the topmost stack frame, and continues to the bottommost frame if necessary.
  ② If a matching handler is found in a stack frame, the VM has to unwind (pop and remove) all those stack frames that are above the matching one; control is then transferred to the execution handler code.

• Possible monitors/locks associated with the removed stack frames will have to be released!!
  – This can get very tricky if there are pending I/O operations associated with the removed stack frames.
Interpreter Performance
Interpretation Overhead

- Interpreted code is generally a lot slower than compiled code/machine code.
  - Studies indicate an order of magnitude difference.
  - Actual range is something like 2.5x to 50x.

- Why?
  - Because there are extra costs associated with interpretation:
    - Dispatch (fetch, decode and invoke) next instruction
    - Access virtual registers and arguments
    - Perform primitive functions outside the interpreter loop

- “Interpreter performance is primarily a function of the interpreter itself and is relatively independent of the application being interpreted.”
Portable vs. Machine-Specific Interpreter Implementations

• In ANSI C, a bytecode interpreter written as a switch statement is usually the fastest choice.
  – There is no other portable way to keep the virtual registers in local variables inside the interpreter loop.
  – GNU C provides additional mechanisms (computed gotos, global register variables) that change the situation.

• In assembly-coded interpreters, direct threaded code is usually the best choice.
  – But only if the underlying hardware can support computed gotos efficiently!

• Assembly-coded interpreters are generally 2-4x faster than fully portable C implementations.
Interpreter Tuning

- Common interpreter optimizations techniques:
  - Writing the interpreter loop and key instructions in assembly code.
  - Keeping the virtual registers (ip, sp, ...) in physical hardware registers – this improves performance dramatically!
  - Splitting commonly used instructions into a separate interpreter loop.
  - Top of stack caching (keeping topmost operand in a register).
  - Padding the instruction lookup table so that it has exactly 16/32/64/128/256 entries.
High-Performance VMs

- Small, simple & portable VM == slow VM
- Interpreted code has a big performance overhead compared to native code:

<table>
<thead>
<tr>
<th>KVM (pure ANSI C)</th>
<th>KVM or HotSpot (with ASM loop)</th>
<th>HotSpot (compiler enabled)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>2-4x</td>
<td>10-20x</td>
</tr>
</tbody>
</table>

- If you need speed, you need a compiler!
  - Unfortunately, compilers are always rather machine/CPU-specific.
  - This introduces a lot of additional complexity.
- Almost always: Fast == more complex
Compilation: Basic Strategies

1. Static / Ahead-Of-Time (AOT) Compilation
   – Compile code before the execution begins.

2. Dynamic (Just-In-Time, JIT) Compilation
   – Compile code on the fly when the VM is running.

• Different flavors of dynamic compilation:
  • Compile everything upon startup (impractical)
  • Compile each method when executed first time
  • Adaptive compilation
Dynamic Compilation Considerations

• **Dynamic compilation must be fast!**
  – Compilation time is taken away from program execution.
  – No time for advanced optimizations; must balance compilation time with execution speed.

• **Machine code takes a lot of space.**
  – Code blow-up ratio (machine code vs. bytecodes) is typically 4-8x.
  – No time or space to compile everything (especially on embedded devices).

• **Solution: Adaptive compilation**
  – Only compile program “hotspots” (frequently executed pieces of code).
  – Analyze code execution on the fly, and tune compilation to match the application needs.
More on High-Performance VMs...

- We will hear more on high-performance virtual machines and adaptive compilation on Oct 29.

<table>
<thead>
<tr>
<th>Class Loader and JAR Reader</th>
<th>Verifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter and Execution Stacks</td>
<td>Compiler (optional)</td>
</tr>
<tr>
<td>Threading System and Thread Scheduler</td>
<td></td>
</tr>
<tr>
<td>Native Interface</td>
<td>Internal Runtime Structures</td>
</tr>
<tr>
<td>Memory System and Garbage Collector</td>
<td></td>
</tr>
</tbody>
</table>
Questions?

“If you don't understand interpreters, you can still write programs; you can even be a competent programmer. But you can't be a master.”

-- D.P. Friedman, M. Wand, T. Haynes, Essentials of Programming Languages