CSc 453
Interpreters & Interpretation

Saumya Debray
The University of Arizona
Tucson

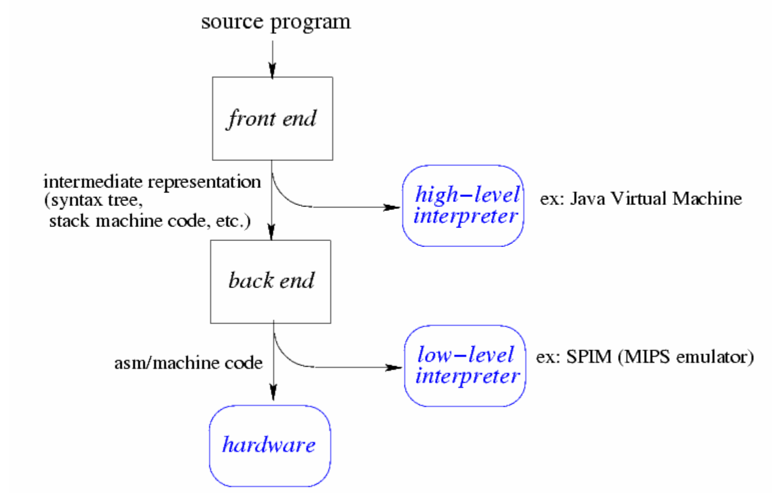
Interpreters

An *interpreter* is a program that executes another program.

- An interpreter implements a *virtual machine*, which may be different from the underlying hardware platform.

  Examples: Java Virtual Machine; VMs for high-level languages such as Scheme, Prolog, Icon, Smalltalk, Perl, Tcl.
- The virtual machine is often at a higher level of semantic abstraction than the native hardware.
  - This can help portability, but affects performance.

Interpretation vs. Compilation



source program

front end

intermediate representation
(syntax tree,
stack machine code, etc.)

high–level interpreter

ex: Java Virtual Machine

back end

asm/machine code

low–level interpreter

ex: SPIM (MIPS emulator)

hardware

Interpreter Operation

```
ip = start of program;
while ( ¬ done ) {
    op = current operation at ip;
    execute code for op on current operands;
    advance ip to next operation;
}
```

Interpreter Design Issues

- Encoding the operation

  I.e., getting from the opcode to the code for that operation ("dispatch"):

  - byte code (e.g., JVM)
  - indirect threaded code
  - direct threaded code.
- Representing operands
  - register machines: operations are performed on a fixed finite set of global locations ("registers") (e.g.: SPIM);
  - stack machines: operations are performed on the top of a stack of operands (e.g.: JVM).

Byte Code

- Operations encoded as small integers (~1 byte).
- The interpreter uses the opcode to index into a table of code addresses:

```
while ( TRUE ) {
  byte op = *ip;
  switch ( op ) {
    case ADD: … perform addition; break;
    case SUB: … perform subtraction; break;
    …
  }
}
```

- Advantages: simple, portable.
- Disadvantages: inefficient.

Direct Threaded Code

- Indexing through a jump table (as with byte code) is expensive.
- *Idea*: Use the address of the code for an operation as the opcode for that operation.
  - *The opcode may no longer fit in a byte.*

```
while ( TRUE ) {
  addr op = *ip;
  goto *op;  /* jump to code for the operation */
}
```

  - More efficient, but the interpreted code is less portable.

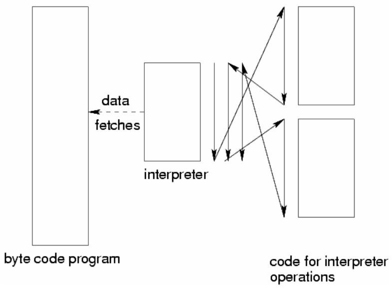
[James R. Bell. Threaded Code. *Communications of the ACM*, vol. 16 no. 6, June 1973, pp. 370–372]

Byte Code vs. Threaded Code

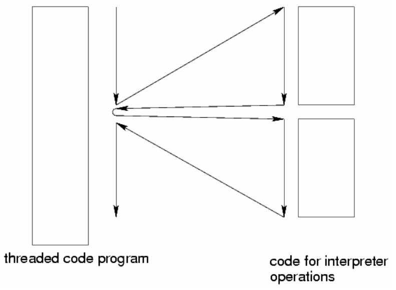
Control flow behavior:

[based on: James R. Bell. Threaded Code. *Communications of the ACM*, vol. 16 no. 6, June 1973, pp. 370–372]

byte code:



threaded code:



Indirect Threaded Code

- Intermediate in portability, efficiency between byte code and direct threaded code.
- Each opcode is the address of a word that contains the address of the code for the operation.
  - *Avoids some of the costs associated with translating a jump table index into a code address.*

```
while ( TRUE ) {
  addr op = *ip;
  goto **op;  /* jump to code for the operation */
}
```

[R. B. K. Dewar. Indirect Threaded Code. *Communications of the ACM*, vol. 18 no. 6, June 1975, pp. 330–331]

Example

program operations

`add`
`mul`
`sub`
`mul`
`add`

memory address	***operation***
10000	**`add`**: …
11000	**`sub`**: …
12000	**`mul`**: …
13000	**`div`**: …
	…

byte code		
	Op Table	
	index	*contents*
`17`	17	**10000**
`23`	18	11000
`18`	23	12000
`23`	27	13000
`17`		

direct threaded
10000
12000
11000
12000
10000

indirect threaded		
	Op Table	
	addr	*contents*
`6000`	6000	**10000**
`6008`	6004	11000
`6004`	6008	12000
`6008`	6012	13000
`6000`		

Handling Operands 1: Stack Machines

- Used by Pascal interpreters (’70s and ’80s); resurrected in the Java Virtual Machine.
- *Basic idea*:
  - operands and results are maintained on a stack;
  - operations pop operands off this stack, push the result back on the stack.
- *Advantages*:
  - simplicity
  - compact code.
- *Disadvantages*:
  - code optimization (e.g., utilizing hardware registers effectively) not easy.

Stack Machine Code

- The code for an operation ‘*op* $x_1, \ldots, x_n$’ is:

  push x_n
  …
  push x_1
  op

- *Example*: JVM code for ‘x = 2∗y – 1’:

```
iconst 1     /* push the integer constant 1 */
iload y      /* push the value of the integer variable y */
iconst 2
imul         /* after this, stack contains: ⟨(2*y), 1⟩ */
isub
istore x    /* pop stack top, store to integer variable x */
```

Generating Stack Machine Code

Essentially just a post-order traversal of the syntax tree:

```
void gencode(struct syntaxTreeNode  ∗tnode )
{
   if ( IsLeaf( tnode ) ) {  … }
   else {
     n = tnode→n_operands;
     for (i = n; i > 0; i--) {
       gencode( tnode→operand[i] );            /* traverse children first */
     }  /* for */
     gen_instr( opcode_table[tnode→op] );   /* then generate code for the node */
   }  /* if [else] */
}
```

Handling Operands 2: Register Machines

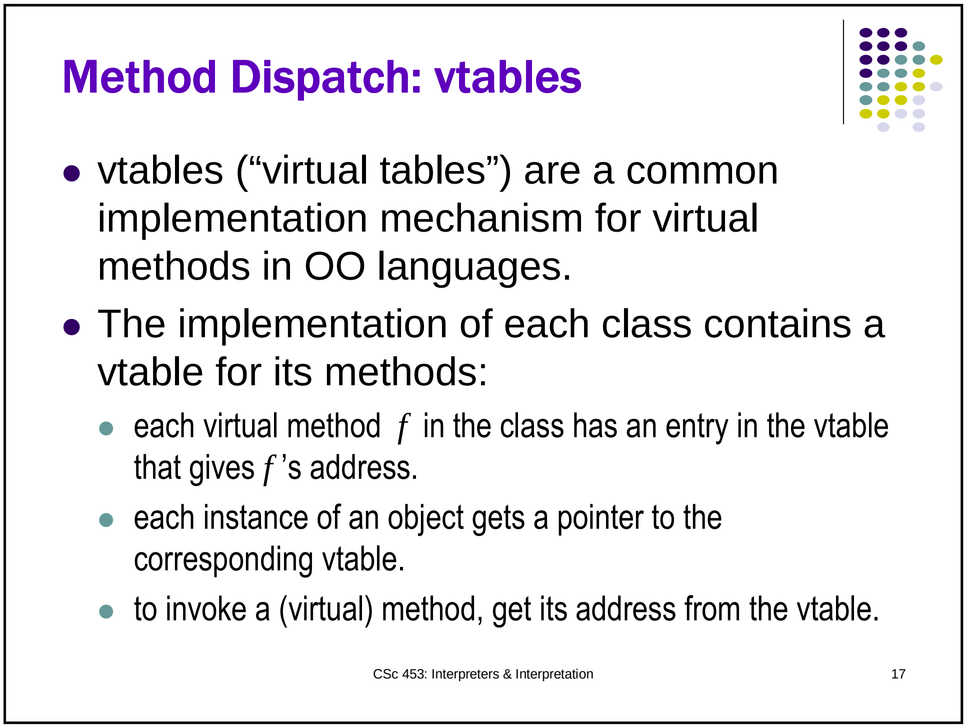
- *Basic idea*:
  - Have a fixed set of "virtual machine registers;"
  - Some of these get mapped to hardware registers.
- *Advantages*:
  - potentially better optimization opportunities.
- *Disadvantages*:
  - code is less compact;
  - interpreter becomes more complex (e.g., to decode VM register names).

Just-in-Time Compilers (JITs)

- *Basic idea*: compile byte code to native code during execution.
- *Advantages*:
  - original (interpreted) program remains portable, compact;
  - the executing program runs faster.
- *Disadvantages*:
  - more complex runtime system;
  - performance may degrade if runtime compilation cost exceeds savings.

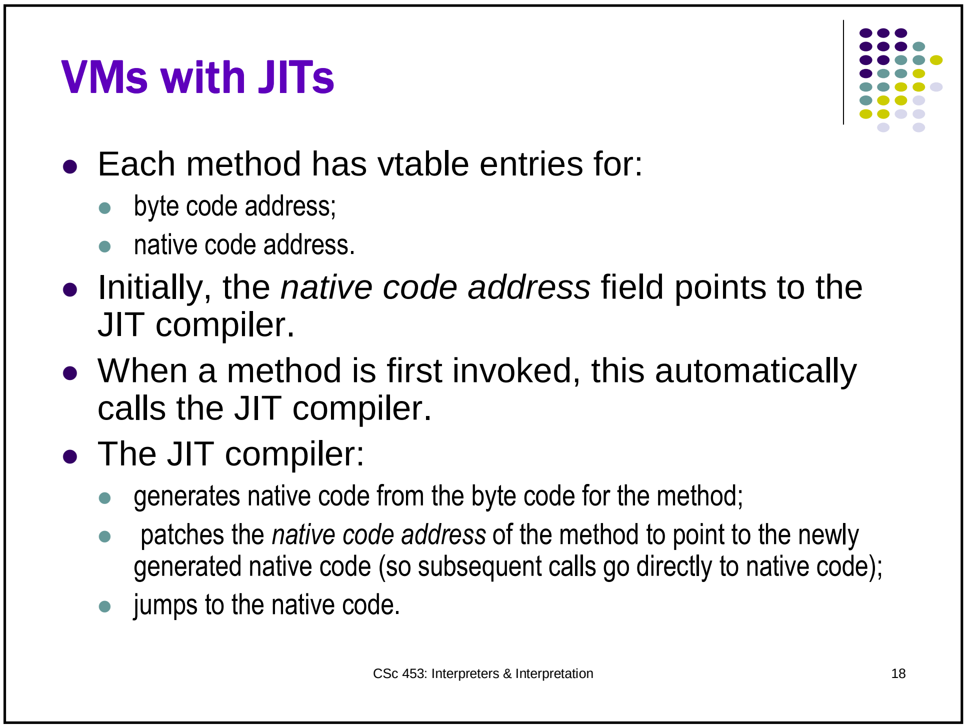
Improving JIT Effectiveness

- *Reducing Costs*:
  - incur compilation cost only when justifiable;
  - invoke JIT compiler on a per-method basis, at the point when a method is invoked.
- *Improving benefits*:
  - some systems monitor the executing code;
  - methods that are executed repeatedly get optimized further.



Method Dispatch: vtables

- vtables (“virtual tables”) are a common implementation mechanism for virtual methods in OO languages.
- The implementation of each class contains a vtable for its methods:
  - each virtual method *f* in the class has an entry in the vtable that gives *f* ’s address.
  - each instance of an object gets a pointer to the corresponding vtable.
  - to invoke a (virtual) method, get its address from the vtable.



VMs with JITs

- Each method has vtable entries for:
  - byte code address;
  - native code address.
- Initially, the *native code address* field points to the JIT compiler.
- When a method is first invoked, this automatically calls the JIT compiler.
- The JIT compiler:
  - generates native code from the byte code for the method;
  - patches the *native code address* of the method to point to the newly generated native code (so subsequent calls go directly to native code);
  - jumps to the native code.

JITs: Deciding what to Compile

- For a JIT to improve performance, the benefit of compiling to native code must offset the cost of doing so.

  E.g., JIT compiling infrequently called straight-line code methods can lead to a slowdown!
- We want to JIT-compile only those methods that contain frequently executed ("hot") code:
  - methods that are called a large number of times; or
  - methods containing loops with large iteration counts.

JITs: Deciding what to Compile

- Identifying frequently called methods:
  - count the number of times each method is invoked;
  - if the count exceeds a threshold, invoke JIT compiler.
  - (In practice, start the count at the threshold value and count down to 0: this is slightly more efficient.)
- Identifying hot loops:
  - modify the interpreter to "snoop" the loop iteration count when it finds a loop, using simple bytecode pattern matching.
  - use the iteration count to adjust the amount by which the invocation count for the method is decremented on each call.

Typical JIT Optimizations

Choose optimizations that are cheap to perform and likely to improve performance, e.g.:

- inlining frequently called methods:

  consider both code size and call frequency in inlining decision.
- exception check elimination:

  eliminate redundant null pointer checks, array bounds checks.
- common subexpression elimination:

  avoid address recomputation, reduce the overhead of array and instance variable access.
- simple, fast register allocation.