CSc 620
Language-based Approaches to System and Software Security
Christian Collberg
September 29, 1999
Compilation
Copyright © 1999 C. Collberg
Multi-Language
Multi-target Compilers

Slide 2–4

Interpretation

An interpreter is like a compiler (it lexes, parses, performs semantic analysis), only

- It generates virtual machine (VM) code rather than native machine code.
- It executes VM instructions rather than native machine code.

Interpreters are

slow Often 10–100 times slower than executing machine code directly.

portable The virtual machine code is not tied to any particular architecture.

Interpreters work well with

very high-level, dynamic languages (APL, Prolog, ICON) where a lot is unknown at compile-time (array bounds, etc).

Slide 2–6

Kinds of Interpreters

"APL/Prolog-style (load-and-go/interactive) interpreter"

"Java-style interpreter"

Just-In-Time (JIT, Dynamic) Compiler
Actions in an Interpreter

- Internally, an interpreter consists of
  1. The interpreter engine, which executes the VM instructions.
  2. Memory for storing user data. Often separated as a heap and a stack.
  3. A stream of VM instructions.

VM Instruction Sets I

Many virtual machine instruction sets (e.g. Java bytecode, Forth) are stack based.

- `add` pop the two top elements off the stack, add them together, and push the result on the stack.
- `push X` push the value of variable X.
- `pusha X` push the address of variable X.
- `store` pop a value V, and an address A off the stack. Store V at memory address A.

<table>
<thead>
<tr>
<th>Source Code</th>
<th>VM Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR X,Y,Z : INTEGER;</td>
<td>pusha X</td>
</tr>
<tr>
<td>BEGIN</td>
<td>push Y</td>
</tr>
<tr>
<td>X := Y + Z;</td>
<td>push Z</td>
</tr>
<tr>
<td>END;</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>store</td>
</tr>
</tbody>
</table>

VM Instruction Sets II

- Stack codes are compact. If we don’t worry about code size, we can use any intermediate code (tuples, trees).

Example: RISC-like VM code with \( \infty \) number of virtual registers \( R_1, \ldots \):

- `add R_1, R_2, R_3` Add VM registers \( R_2 \) and \( R_3 \) and store in VM register \( R_1 \).
- `load R_1, X` \( R_1 \) := value of variable \( X \).
- `loada R_1, X` \( R_1 \) := address of variable \( X \).
- `store R_1, R_2` Store value \( R_2 \) at address \( R_1 \).

<table>
<thead>
<tr>
<th>Source Code</th>
<th>VM Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR X,Y,Z : INTEGER;</td>
<td>[1] pusha X</td>
</tr>
<tr>
<td>BEGIN</td>
<td>[2] push 1</td>
</tr>
<tr>
<td>X := 1;</td>
<td>[3] store</td>
</tr>
<tr>
<td>WHILE X &lt; 10 DO</td>
<td>[4] push X</td>
</tr>
<tr>
<td></td>
<td>[5] push 10</td>
</tr>
<tr>
<td></td>
<td>[6] GE</td>
</tr>
<tr>
<td></td>
<td>[7] BrTrue 14</td>
</tr>
<tr>
<td></td>
<td>[8] pusha X</td>
</tr>
<tr>
<td></td>
<td>[9] push Y</td>
</tr>
<tr>
<td></td>
<td>[10] push Z</td>
</tr>
<tr>
<td></td>
<td>[12] store</td>
</tr>
<tr>
<td></td>
<td>[13] jump 4</td>
</tr>
<tr>
<td>ENDDO</td>
<td></td>
</tr>
<tr>
<td>END;</td>
<td></td>
</tr>
</tbody>
</table>
Switch Threading I

- Instructions are stored as an array of integer tokens. A switch selects the right code for each instruction.

```c
typedef enum {add, load, store, ...} Inst;
void engine () {
    static Inst prog[] = {load, add, ...};

    Inst *pc = &prog;
    Stack int[100];
    int sp = 0;

    for (; ;)
        switch (*pc++) {
        case add:
            Stack[sp-1]=Stack[sp-1]+Stack[sp];
            sp--; break;
        }
```

Switch Threading II

- Switch (case) statements are implemented as indirect jumps through an array of label addresses (a jump table). Every switch does 1 range check, 1 table lookup, and 1 jump.

```c
JumpTab = [0, Lab1, Lab2, &Lab3, &Lab2];

switch (c) {
  case 1: S1; break; // goto JumpTab[c];
  case 2: S2; break; // goto JumpTab[c];
  case 3: S3; break; => Lab1: S1; goto Lab4;
  default: S4; goto Lab4;
}
```
Instruction Sets Revisited

- We can (sometimes) speed up the interpreter by being clever when we design the VM instruction set:
  1. Combine often used code sequences into one instruction. E.g.,
     \[ \text{muladd } a, b, c, d \text{ for } a := b \cdot c + d. \]
     This will reduce the number of instructions executed, but will make the VM engine larger.
  2. Reduce the total number of instructions, by making them simple and RISC-like. This will increase the number of instructions executed, but will make the VM engine smaller.
- A small VM engine may fit better in the cache than a large one, and hence yield better overall performance.

Just-In-Time Compilation

- Used to be called Dynamic Compilation before the marketing department got their hands on it. Also a verb, jitting.
- The VM code is compiled to native code just prior to execution. Gives machine independence (the bytecode can be sent over the net) and speed.
- When? When a class/module is loaded? The first time a method/procedure is called? The 2nd time it’s called?