Interpreters

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.).
### 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.

```
add
store
mul
....
....
```

**Memory**

```
Stack

Heap

"Hello!"
Static Data
```

---

### 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; BEGIN X := Y + Z; END;</td>
<td>pusha X push Y push Z add store</td>
</tr>
<tr>
<td>ENDDO</td>
<td>[13] jump 4</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 \( R_1, \cdots \):

- **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; BEGIN X := Y + Z; END;</td>
<td>load ( R_1, Y ) load ( R_2, Z ) add ( R_3, R_1, R_2 ) loada ( R_4, X ) store ( R_4, R_3 )</td>
</tr>
</tbody>
</table>

---

### Stack Machine Example I

<table>
<thead>
<tr>
<th>Source Code</th>
<th>VM Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>END;</td>
<td>[13] jump 4</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.

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
switch (c) {
    case 1: S1; break;
    case 2: if ((c < 1) || (c > 3)) goto Lab2;
             goto *jumpTab[c];
    case 3: S2; break;
    default: S3; goto Lab4;
}
```

JumpTab = {0, &Lab1, &Lab2, ..., &Lab4};
Direct Call Threading I

- Every instruction is a separate function.
- The program is an array of pointers to these functions. i.e. the add instruction is represented as the address of the add function.

```c
typedef void (* Inst)();
Inst prog[] = {&load, &add, ...};
```

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

void add(); {
    Stack[sp-1]=Stack[sp-1]+Stack[sp];
    sp--;
}

void engine () { for (; ;) (*pc++)(); }
```

Direct Call Threading III

- In direct call threading all instructions are in their own functions.
- This means that VM registers (such as pc, sp) must be in global variables.
- So, every time we access pc or sp we have to load them from global memory. ⇒ Slow.
- With the switch method pc and sp are local variables. Most compilers will keep them in memory. ⇒ Faster.
- Also, a direct call threaded program will be large since each instruction is represented as a 32/64-bit address.
- Also, overhead from call/return sequence.

Direct Threading I

- Each instruction is represented by the address (label) of the code that implements it. At the end of each piece of code is an indirect jump to the next instruction.
- 
  "&&" takes the address of a label. goto *V jumps to the label whose address is stored in variable V. This is a gcc extensions to C.

```c
typedef void *Inst;
void engine() {
    static Inst prog[]={&add, &store, ...};
    Inst *pc = &prog; int sp;
goto *pc++;
add:   Stack[sp-1]=Stack[sp-1]+Stack[sp];
        sp--; goto *pc++;
}
Indirect Threading I

- Unfortunately, a direct threaded program will be large since each instruction is an address (32 or 64 bits).
- At the cost of an extra indirection, we can use byte-code instructions instead:

```c
typedef enum {add, load, store, …} Inst;
typedef void *Addr;

void engine() {
    static Inst prog[]={add, load, …};
    static Addr jtab[]={&add, &load, …};
    Inst *pc = &prog; int sp;
goto *jtab[(*pc++)];
}
```

add:
Stack[sp-1]=Stack[sp-1]+Stack[sp];
sp--; goto *jtab[(*pc++)];

Minimizing Stack Accesses

- To reduce the cost of stack manipulation we can keep one or more of the Top-Of-Stack elements in registers.
- In the example below, TOS holds the top stack element. Stack[sp] holds the element second to the top, etc.

```c
void engine() {
    static Inst prog[]={&add, &store, …};
    Inst *pc = &prog; int sp;
    goto *pc++;
    add: TOS= TOS+ Stack[sp];
        sp--; goto *pc++;
    store: Memory[Stack[sp]]=TOS;
        TOS= Stack[sp-1]; sp--; goto *pc++;
}
```
**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., `muladd a, b, c, d` for `a := b * 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.

**Superoperators I**

- Proebsting’s hti C interpreter uses superoperators to speed up interpretation.
- The idea is simple:
  1. Compile a C program to intermediate code,
  2. infer new operators for common idioms,
  3. generate a specialized interpreter for the inferred bytecode.
- hti uses lcc (a free C compiler) which generates expression trees. lcc generates only 109 different intermediate code instructions. This leaves us 147 codes to play with. If `x->b` is a very common operation in the generated code, emit a special bytecode for this operation.

**Superoperators II**

```
C source code
  ↓
  lcc C front-end
  ↓
  Superoperator analysis
  ↓
  Intermediate code (max 109 operators)
  ↓
  Intermediate code (max 256 operators)
  ↓
  Interpreter generation
  ↓
  Bytecode generation
  ↓
  Bytecode + Interpreter
```

**Superoperators III**

- Without superoperators hti runs 8-16 times slower than unoptimized native code. With superoperators it’s only slower by a factor of 3-9.
- The optimized code is smaller, since complex expressions such as `a=a+1` which would previously have generated `pusha; push a; push 1; add; store` now might only generate `incr a 1`.
- The optimized interpreter is larger since it has to handle more bytecodes.
- If some types of operations (e.g. floating point) are not used at all in a particular program these instructions can be eliminated resulting in a smaller interpreter, or at least reused for more superoperators.
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?

```
Java Source
ALPHA
\[\rightarrow\]
Java compiler
Server
Transfer bytecodes over the net
byte-code
SPARC
\[\rightarrow\]
Java VM interpreter
\[\rightarrow\]
Java bytecode to native JIT
\[\rightarrow\]
Native machine code
```

Readings and References


Summary I

- Direct threading is the most efficient dispatch method. It cannot be implemented in ANSI C. Gnu C’s “labels as values” do the trick.
- Indirect threading is almost as fast as direct threading. It may sometimes even be faster, since the interpreted program is smaller and may hence fits better in the cache.
- Call threading is the slowest method. There is overhead from the jump, save/restore of registers, the return, as well as the fact that VM registers have to be global.

Summary II

- Switch threading is slow but has the advantage to work in all languages with a case statement.
- The interpretation overhead consists of *dispatch overhead* (the cost of fetching, decoding, and starting the next instruction) and *argument access overhead*.
- You can get rid of some of the argument access overhead by *caching* the top \( k \) elements of the stack in registers. See Ertl’s article.
- Jitting is difficult on machines with separate data and code caches. We must generate code into the data cache, then do a *cache flush*, then jump into the new code. Without the flush we’d be loading the old data into the code cache!