

CS577 Modern Language Processors

Spring 2008

Lecture 2

MAKING INTERPRETERS EFFICIENT

- VM programs have an explicitly specified binary representation, typically called **bytecode**.
- Most VM's can execute bytecode directly by **interpretation**.
- Interpretation is typically 1-2 orders of magnitude slower than compilation (but of course this depends on interpreter, compiler, target machine)
- So serious VM's usually do JIT compilation too
- Still, it is worthwhile to make interpreters **efficient**
- But it is also desirable to keep them **portable**

TARGET HARDWARE: FACTS OF LIFE

Accessing memory is **slow!**

- Even a L1 cache hit typically costs several cycles.
- Cache misses cost 10's-100's of cycles.
- Must try to keep data in registers if possible.
- Small changes in data, code layout can have big effects

Machines are deeply pipelined!

- Conditional branches are bad.
- Dynamic-target branches are worse.
- Should try to utilize hardware support tricks (e.g. branch target buffers).

IA32 (X86, Pentium)

- Small number of registers (effectively 6, with some special restrictions)
- Complex instruction set (many addressing modes)
- Lots of optimization (e.g., superscalar issue, out-of-order issue, register shadowing) done in hardware

SPARC (and other RISC machines)

- Lots of registers (about 30)
- Very simple instruction set
- Instruction latencies are exposed, so software scheduling can help

PowerPC is somewhere in-between.

Other processors to remember

- low power/embedded processors
- "EPIC" (explicitly parallel instruction computing, e.g IA64)
- multicore (shared-memory multiprocessor)

COSTS OF INTERPRETATION

Interpreting an instruction requires:

- Dispatching the instruction: getting control to the code corresponding to the instruction
- Accessing the operands: getting the values of the parameters and arguments (and storing the result)
- Actually performing the computation. (Note: interpreters win more when this is slow!)

NAIVE INTERPRETER: SAMPLE INSTRUCTIONS

```

u4 stack[STACKSIZE];
u4* sp;
void interp (Method *method) {
    u1 *pc = method->code;
    while (1) {
        switch (*pc) {
            case ICONST_0:
                { *(++sp) = (u4) 0;
                  pc++;
                  break; }
            case ISTORE_0:
                { locals[0] = *(sp--);
                  pc++;
                  break; }
            case IADD:
                { int32 v2 = (int32) (*(sp--));
                  int32 v1 = (int32) (*sp);
                  *sp = (u4) (v1 + v2);
                  pc++;
                  break; }
            ...
        }
    }
}

```

SPEEDING UP OPERAND ACCESS

First, let's consider just the cost of accessing stack elements: loads/stores to memory and sp adjustment.

C code:

```
case ICONST_0: { *(++sp) = (u4) 0; pc++; break; }
```

SPARC machine code (obtained using gcc -S)

```
; initially %l2 = %hi(sp)    %l1 = pc
ld          [%l2+%lo(sp)], %g1    ; load value of sp from C global
add        %l1, 1, %l1          ; pc++
add        %g1, 4, %o5          ; new sp
st         %g0, [%g1+4]        ; *(new sp) = 0    (%g0 is always 0)
ba,pt     %xcc, top            ; (unconditional branch) break
st         %o5, [%l2+%lo(sp)]   ; DELAY SLOT!: save new sp into C global
```

HELPING THE REGISTER ALLOCATOR

Keeping `sp` in a global memory location looks like a terrible idea, since it requires one load and one store per bytecode executed.

Let's make it a local of `interp` instead. Then C compiler should be able to keep it in a register (if there are any available).

New SPARC machine code:

```
; initially %i2 = sp %l1 = pc
add    %i2, 4, %i2      ; ++sp
add    %l1, 1, %l1     ; pc++
ba,pt  %xcc, top       ; break
st     %g0, [%i2]      ; DELAY SLOT!: *(sp) = 0
```

TESTING THE IMPROVEMENT

Let's check if this change speeds up the interpreter.

- To do this properly, we need a benchmark suite.
- But we'll do something really simple (`Sum.java`)

Some Performance Results

| Machine | sp global | sp local |
|--------------------|-----------|----------|
| UltraSPARC3 1+?GHz | 27s | 26s |
| Pentium4 1.4GHz | 29 | 27 |
| PentiumIII 1GHz | 35 | 36 |
| Core Duo 2.33GHz | 15 | 13.5 |
| PowerPC 1.5GHz | 17 | 14 |

(Should do multiple runs, especially on shared machines.)

Why aren't improvements bigger on SPARC? What about PentiumIII??

MORE CODE WITH LOCAL sp

C code:

```
case ISTORE_0: { locals[0] = *(sp--); pc++; break;}
```

SPARC code:

```
; initially %i2 = sp      %l1 = pc      %l2 = base of locals
ld      [%i2], %g1      ; %g1 = *sp
add     %l1, 1, %l1      ; pc++
add     %i2, -4, %i2     ; sp--
ba,pt   %xcc, top       ; break
st      %g1, [%l2]      ; DELAY SLOT!: locals[0] = *sp
```

STILL MORE CODE

C code:

```
case IADD: { int32 v2 = (int32) (*(sp--));
            int32 v1 = (int32) (*sp);
            *sp = (u4) (v1 + v2);
            pc++; break; }
```

SPARC code:

```
; initially %i2 = sp      %l1 = pc
ld      [%i2], %o5      ; *sp
add     %i2, -4, %i2    ; sp--
add     %l1, 1, %l1     ; pc++
ld      [%i2], %g1      ; *(new sp)
add     %g1, %o5, %g1   ; %g1 = computed result
ba,pt  %xcc, top       ; break
st      %g1, [%i2]     ; DELAY SLOT!: *(new sp) = result
```

Next obvious problem is that nearly every instruction loads and/or stores stack entries.

STACK CACHING

Idea: what if we **cache** the top-of-stack in a local variable `s0`?

(Assume that `sp` points to the top of the *remainder* of the stack.)

This saves one load and one store for `IADD`:

```
case IADD: {int32 v2 = (int32) s0; int32 v1 = (int32) (*(sp--));
           s0 = (u4) (v1+v2); pc++; break; }
```

Approximate SPARC code:

```
; initially %i2 = sp      %i3 = s0      %l1 = pc
ld          [%i2], %o5    ; *sp
add         %i2, -4, %i2  ; sp--
add         %i3, %o5, %i3 ; s0 = *sp + s0
ba,pt      %xcc, top     ; break
add        %l1, 1, %l1   ; DELAY SLOT!: pc++
```

CACHING ONE SLOT

But it is a wash for the other two instructions because we have to keep `s0` up-to-date.

```
case ICONST_0: { *(++sp) = s0; s0 = (u4) 0; pc++; break; }
```

Approximate SPARC code (still one store)

```
; initially %i2 = sp %i3 = s0 %l1 = pc
add    %i2, 4, %i2      ; ++sp
st     %i3, [%i2]      ; *(sp) = s0
add    %l1, 1, %l1     ; pc++
ba,pt  %xcc, top       ; break
mov    %g0, %i3        ; DELAY SLOT!: s0 = 0
```

CACHING ONE SLOT (CONTINUED)

```
case ISTORE_0: { locals[0] = s0; s0 = *(sp--); pc++; break; }
```

Approximate SPARC code (still one load and one store)

```
; initially %i2 = sp    %i3 = s0    %l1 = pc    %l2 = base of locals
st      %i3, [%l2]      ; locals[0] = s0
add     %l1, 1, %l1     ; pc++
ld      [%i2], %i3      ; s0 = *sp
ba,pt   %xcc, top       ; break
add     %i2, -4, %i2    ; DELAY SLOT!: sp--
```

CACHING TWO SLOTS

What if we keep **two** elements in local variables (registers) named `s1` (top of stack) and `s0` (next-to-top of stack)?

```
case ISTORE_0: { locals[0] = s1; s1 = s0; s0 = *(sp--);  
                pc++; break; }
```

```
case ICONST_0: { *(++sp) = s0; s0 = s1; s1 = (u4) 0;  
                pc++; break; }
```

```
case IADD: { int32 v2 = (int32) s1; int32 v1 = (int32) s0;  
            s1 = (u4) (v1+v2); s0 = *(sp--); pc++; break; }
```

This just pushes off the problem: no improvement in number of loads and stores needed.

New idea: let's keep a **different** number of cached stack slots at different points during execution.

GENERALIZED STACK CACHING

- Interpreter operates in one several different **states** corresponding to how many stack slots are cached.
- Each instruction (potentially) causes transition to a different state, according to what it does to the stack.
- For example:

ICONST_0 moves to a state where *more* slots are cached;

ISTORE_0 moves to one where *fewer* slots are cached.

IADD moves to a state where one slot is cached.

GENERALIZED STACK CACHING (2)

For JVM, 3 states are sufficient.

State 0: no slots cached.

State 1: top of stack is cached in variable $s0$.

State 2: top of stack is cached in variable $s1$; next-to-top in $s0$.

In all states, sp points to remainder of stack beyond cached slots.

Sample code follows (in practice we may organize it differently)...

```

case IADD: {
    switch (state) {
    case 0: { int32 v2 = (int32) *(sp--); int32 v1 = (int32) *(sp--);
            s0 = (u4) (v1+v2); state = 1; break; }
    case 1: { int 32 v2 = (int32) s0; int32 v1 = (int32) *(sp--);
            s0 = (u4) (v1+v2); state = 1; break; }
    case 2: { int 32 v2 = (int32) s1; int32 v1 = (int32) s0;
            s0 = (u4) (v1+v2); state = 1; break; }
    pc++; break; }

case ICONST_0: {
    switch (state) {
    case 0: s0 = 0; state = 1; break;
    case 1: s1 = 0; state = 2; break;
    case 2: *(++sp) = s0; s0 = s1; s1 = 0; state = 2; break; }
    pc++; break; }

case ISTORE_0: {
    switch (state) {
    case 0: locals[0] = *(sp--); state = 0; break;
    case 1: locals[0] = s0; state = 0; break;
    case 2: locals[0] = s1; state = 1; break; }
    pc++; break; }

```

EXAMPLE SEQUENCE

Consider a typical expression like

$$b = a + 3$$

where we assume a is local variable 0 and b is local variable 1.

(Assume we start with state = 0.)

| Bytecode | Corresponding executed code |
|----------|---|
| ILOAD_0 | <code>s0 = locals[0]; state = 1;</code> |
| ICONST_3 | <code>s1 = 3; state = 2;</code> |
| IADD | <code>s0 = s1 + s0; state = 1;</code> |
| ISTORE_1 | <code>locals[1] = s0; state = 0;</code> |

We do only the essential loads and stores – no stack traffic at all!

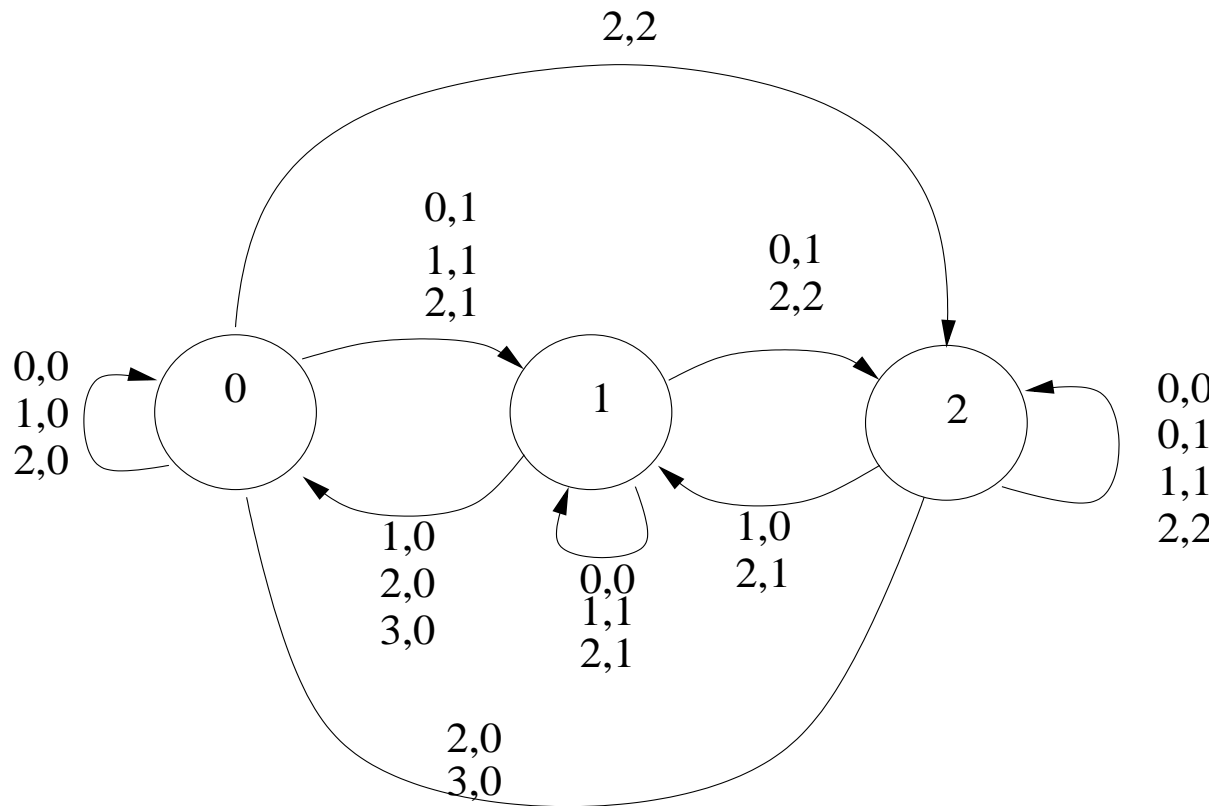
3-STATE TRANSITION DIAGRAM

More generally, instructions are classified by a pair:

(# of stack slots they consume, # of stack slots they produce)

For example:

| | |
|----------|-----|
| ISTORE_0 | 1,0 |
| ICONST_0 | 0,1 |
| IADD | 2,1 |



DYNAMIC VS. STATIC CACHING

So far we've described **dynamic** stack caching, where the interpreter keeps track of its current state.

- In practice, we implement this by having three complete sets of instruction implementations and dispatching to the correct one based on current state as well as opcode (more on this later).
- But it may seem like we should be able to predict the state at each program point **statically** (before execution). If so, we could simply have three variants of each opcode, and select the right one at compile time. This would be more efficient.
- Only problem: at **join points** in the code, the state may differ depending on the path by which the join point was reached. Must choose a convention for which state to use there, and add **compensation code** to the other branches; this is complex in practice.

ASIDE: WHY USE STACK-BASED VM'S?

Nearly all hardware processors use **registers** (although some early architectures were stack-based).

- Each HW instruction is parameterized by its argument/result registers.
- Why is this good for hardware? Because the opcode and the argument registers can be decoded in **parallel**, and values can quickly be fetched from a small, fast register file.

Why not try this in software machines too?

- Parameters must be fetched from the byte stream and decoded **serially**; stack instructions, whose parameters are implicit, don't require this.
- Software registers cannot easily be stored in hardware registers, because the latter can't be indexed. So software registers end up living in an in-memory array.
- On the other hand, register architectures require fewer instructions; hence less **dispatch**. So maybe a worthwhile idea after all...

SPEEDING UP INSTRUCTION DISPATCH

What does SPARC code look like now?

```
        ; %14 = table    %11 = pc
top:
    ldub    [%11], %o1    ; *pc
    and     %o1, 0xff, %g1 ; (u1) *pc
    cmp     %g1, 200      ; if >200
    bgu,pn %icc, undefined ; or <0, branch
    sll     %g1, 2, %g1   ; scale
    ld      [%14+%g1], %o4 ; fetch snippet address
    jmp     %o4           ; jump to snippet
    nop
table:
    .word  nop_snippet
    .word  aconst_null_snippet
    .word  iconst_m1_snippet
    .word  iconst_0_snippet
    ...
    .word  goto_w_snippet
undefined:
    ...issue error and die...
```

THEADED CODE

Obvious performance problems:

- Unnecessary bounds check.
- Two jumps per dispatch (counting the one back to `top` at the end of the previous instruction).

First fix: **(Indirect) Threaded Code**

If we can code our own indirect jumps, could

- Remove bounds check.
- Replicate dispatch at end of every snippet, thus removing one jump.
- This is not possible in ANSI Standard C, but can do in `gcc` using the `&&` operator.

INDIRECT THREADED CODE

```
interp(Method method) {
    static void *dispatch_table[] =
        {&&NOP,
         &&ACONST_NULL,
         &&ICONST_M1,
         ...,
         &&JSR_W };
    u1 *pc = method->code;
    ...
    goto *(dispatch_table[*pc]);

NOP:
    pc++;
    goto *(dispatch_table[*pc]);

ACONST_NULL:
    *(++sp) = (u4) 0;
    pc++;
    goto *(dispatch_table[*pc]);
    ...
}
```

USING PROCESSOR BRANCH SUPPORT

One extra reason why indirect threaded code improves performance may be that it makes (slightly) better use of hardware support for branch predication.

Many pipelined processors contain a **branch target buffer** (BTB) that dynamically remembers the last target for each (static) conditional or indirect branch instruction. The next time the branch is executed, the processor pre-fetches from the address predicted by the buffer.

- A naive interpreter makes **terrible** use of this feature, because a **single** instruction dispatches to all the snippets, so the prediction accuracy is 0.
- The indirect threaded code version does somewhat better, because the dispatches are distributed, and certain bytecode instruction sequences are quite common, so prediction accuracy may be > 0 .

But a fundamental prediction mismatch between the VM and the target hardware remains.

DIRECT THREADING

Each instruction dispatch still requires two fetches: one to get the byte code and a second to get the snippet address.

New idea: what if we represent each instruction opcode by the address of its snippet?

```
interp() {
    u4 codeaddrs[] = ...; /* fill this with snippet addrs */
    u4 *pc = codeaddrs; /* initialize to start */
    goto **pc;

    ACONST_NULL:
        *(++sp) = (u4) 0;
        pc++;
        goto **pc;
    ...
}
```

Now need only one fetch per instruction!

REWRITING BYTE CODE

But notice that we're no longer interpreting the original bytecode any more.

Must rewrite before execution

Simple in principle, but there are details. e.g.

- What should we do with the parameter bytes following the opcode?

If we're going to rewrite the bytecode, there are **many** opportunities to improve things, e.g.

- Combine code for similar opcodes (e.g. constant loading).
- Short-circuit constant pool references (important in full language)
- Perform static stack caching
- Etc, etc.

A more radical rewrite idea: put each snippet in a separate **subroutine** and replace instruction sequences by series of subroutine calls. May pay off on processors that pre-fetch from the return address on the hardware stack! (Also more portable than computed goto.)

REDUCING DISPATCHES

Another way to reduce dispatch time is to do **fewer** dispatches.

One basic approach is to **combine** sequences of instructions that occur frequently into into “macro” or “super”-instructions.

For example, the following sequence pattern is very common:

```
ILOAD n  
ICONST i  
IADD  
ISTORE n
```

In fact, the JVM designers already invented a combined instruction for this (IINC) but the same idea works for other sequences.

Another approach is to use a **register** architecture, which typically requires many fewer instructions (although each instruction gets more parameters).

BUILDING COMBINED INSTRUCTIONS

This can be done in several ways:

- Statically, for multiple programs:
 - Essentially a refinement of the VM definition, possibly tuned to workload from a particular set of programs.
 - Can construct such specialized VM's semi-automatically from a generic VM.
 - Specialized VM can be compiled with “cross-snippet” optimization.
- Statically, for a single program
 - Encoding is sent with the program.

Static encodings also have the benefit of reducing the program size, allowing quicker transmission.

- Dynamically, by building superinstructions “on the fly” from snippet code.
 - This is beginning to resemble a compiler!