CS4/510 Compilers and Interpreters
Spring 2021
Lecture on Interpreters
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** (e.g. stick to standard C)
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 probably worse.
- Should try to utilize hardware support tricks (e.g. branch target buffers).
- Recent machines (last 10 years) may have significantly improved branch prediction.
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: the longer this takes, the smaller the percentage overhead of interpretation!)
uintptr_t stack[STACKSIZE];
void interp (Method *method,uintptr_t *sp) {
    char *pc = method->code;
    uintptr_t *locals = sp - method->nargs + 1;
    sp = locals + method->max_locals - 1;
    while (1) {
        switch (*pc) {
        case ICONST_3:
            *(++sp) = (uintptr_t) 3;
            pc++; 
            break;  
        case ISTORE_1:
            locals[1] = *(sp--);
            pc++; 
            break;  
        case IADD:
            {  
            int32_t v2 = (int32_t) (*sp--);
            int32_t v1 = (int32_t) (*sp);
            *sp = (uintptr_t) (v1 + v2);
            pc++; 
            break;  
            }
        ... 
        }
    }}}}
First, let’s consider just the cost of accessing stack elements: loads/stores to memory and \( sp \) adjustment.

C code:

```c
    case ICONST_3: { *(++sp) = (uintptr_t) 3; pc++; break; }
```

X86 (64-bit) machine code (obtained using clang -S)

```assembly
    // %rbx holds sp; %r14 holds pc
    movq $3, 8(%rbx) // *(new sp) = 3
    addq $8, %rbx // new sp = sp + 8
    incq %r14 // pc++
    jmp top
```

This code is pretty tight, assuming that the stack must be held in memory.
C code:

```c
case ISTORE_1: { locals[1] = *(sp--); pc++; break; }
```

X86 (64-bit) machine code (obtained using clang -S)

```
// %rbx holds sp; %r14 holds pc; %r13 points to base of locals
movq (%rbx), %rax  // *old_sp
addq $-8, %rbx     // new sp = sp - 8
movq %rax, 8(%r13) // locals[1] = *old_sp
incq %r14          // pc++
jmp top
```

Again, hard to do better, assuming that locals are held in memory.
C code:

```c
    case IADD: { int32_t v2 = (int32_t) (*(sp--));
                 int32_t v1 = (int32_t) (*sp);
                 *sp = (u4) (v1 + v2);
                 pc++; break; }
```

X86-64 code:

```assembly
    // %rbx holds sp; %r14 holds pc
    movl  -8(%rbx), %eax       // *(sp--)
    addl  (%rbx), %eax         // *sp
    cltq
    movq  %rax, -8(%rbx)       // store to *(new_sp)
    leaq  -8(%rbx), %rbx      // new_sp = sp - 8
    incq  %r14                 // pc++
```

Most obvious problem is that nearly every instruction loads and/or stores stack entries.
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_t v2 = (int32_t) s0;
    int32_t v1 = (int32_t) (*(sp--));
    s0 = (uintptr_t) (v1+v2); pc++; break; }
```

Approximate X86-64 code:

```
// %rbx holds sp (pointer to slot1); %r14 holds pc;
// %r10 holds slot0
movl  (%rbx), %eax  // load *sp
cltq    // sign extend %eax into %rax
addq  %rax, %r10  // slot0 = *sp + slot0
leaq  -8(%rbx), %rbx  // new_sp = sp - 8
incq  %r14  // pc++
jmp   top
```
**CACHING ONE SLOT**

But it is a wash for the other two instructions because we have to keep \( s_0 \) up-to-date.

```c
  case ICONST_3: { *(++sp) = s0; s0 = (u4) 0; pc++; break; }
```

Approximate X86-64 code (still one store)

```assembly
  // %rbx holds sp (pointer to slot1); %r14 holds pc;
  // %r10 holds slot0
  leaq  8(%rbx), %rbx   // new_sp = sp + 8
  movq  %r10, (%rbx)    // *new_sp = slot0
  movq  $3, %r10        // slot0 = 3
  incq  %r14            // pc++
  jmp   top
```
CACHING ONE SLOT (CONTINUED)

```java
        case ISTORE_1: { locals[1] = s0; s0 = *(sp--); pc++; break; }
```

Approximate X86-64 code (still one store)

```
        // %rbx holds sp (pointer to slot1);
        // %r13 points to base of locals
        // %r14 holds pc; %r10 holds slot0
        movq %r10, 0(%r13)  // locals[0] = slot0
        movq (%rbx), %r10   // slot0 = *sp
        leaq -8(%rbx), %rbx // sp = sp - 8
        incq %r14            // pc++
        jmp top
```
What if we keep two elements in local variables (registers) named \(s_1\) (top of stack) and \(s_0\) (next-to-top of stack)?

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

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

```java
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.
For JVM, 3 states are sufficient to handle all instruction types.

State 0: no slots cached.

State 1: top of stack is cached in variable $s_0$.

State 2: top of stack is cached in variable $s_1$; next-to-top in $s_0$.

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: { int32 v2 = (int32) s0; int32 v1 = (int32) (*(sp--));
        s0 = (u4) (v1+v2); state = 1; break; }
    case 2: { int32 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; }
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.)

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Corresponding executed code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILOAD_0</td>
<td>s0 = locals[0]; state = 1;</td>
</tr>
<tr>
<td>ICONST_3</td>
<td>s1 = 3; state = 2;</td>
</tr>
<tr>
<td>IADD</td>
<td>s0 = s1 + s0; state = 1;</td>
</tr>
<tr>
<td>ISTORE_1</td>
<td>locals[1] = s0; state = 0;</td>
</tr>
</tbody>
</table>

We do only the essential loads and stores – no stack traffic at all!
More generally, instructions are classified by a pair:

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

For example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SLOTS CONSUMED</th>
<th>SLOTS PRODUCED</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISTORE_0</td>
<td>1,0</td>
<td>0,0</td>
</tr>
<tr>
<td>ICONST_0</td>
<td>0,1</td>
<td>1,0</td>
</tr>
<tr>
<td>IADD</td>
<td>2,1</td>
<td>2,0</td>
</tr>
</tbody>
</table>

3-State Transition Diagram
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.
Nearly all hardware processors use registers

- 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; for stack instructions, parameters are implicit.
- Instructions with parameters take more space.
- 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 (just like stack slots).
- On the other hand, register architectures require fewer instructions; hence less dispatch. So maybe a worthwhile idea after all...
What does X86-64 code look like now?

// %r12 holds table; %r14 holds pc

```assembly
top:  movzbl (%r14), %rax       // fetch opcode at pc
cmpq $tablesizel, %rax        // compare against jump table size
ja  undefined                 // if out of range branch to "undefined"
movslq (%r12,%rax,4), %rax   // get table entry=snippet address-table base
addq %12,%rax                 // add to table base
jmpq *%rax                     // jump to snippet
```

table:

```assembly
    .word nop_snippet-table
    .word aconst_null_snippet-table
    .word iconst_m1_snippet-table
    .word iconst_0_snippet-table
    ...
    .word goto_w_snippet-table
```

undefined:

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

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

    ACONST_NULL:
    *(++sp) = (u4) 0;
    pc++;
    goto *(dispatch_table[*pc]);
    ...
}
One extra reason why indirect threaded code improves performance may be that it makes 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 branch instruction (including conditional and indirect branches). The next time the branch instruction 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 prediction accuracy is \( \approx 0 \).
- 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?

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

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

Now need only one fetch per instruction!
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: dispatch to each snippet using a **subroutine** call instruction. May pay off on processors that pre-fetch from the return address on the hardware stack!
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).
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!