Lecture 20: Registers, Functions, Parameters

Three-Address Code to ia32

- The problem is that in reality, the ia32 architecture has very few registers, and example from last lecture used registers profligately.

- *Register allocation* is the general term for assigning virtual registers to real registers or memory locations.

- When we run out of real registers, we *spill* values into memory locations reserved for them.

- We keep a register or two around as *compiler temporaries* for cases where the instruction set doesn't let us just combine operands directly.

A Simple Strategy: Local Register Allocation

- It's convenient to handle register allocation within *basic blocks*—sequences of code with one entry point at the top and (at most) one branch at the end.

- At the end of each such block, spill any registers needed.

- To do this efficiently, need to know when a register is *dead*—that is, when its value is no longer needed.

- We'll talk about how to compute that in a later lecture. Let's assume we know it for now.

- Let's also assume that each virtual register representing a local variable or intermediate result has a memory location suitable for spilling.

Simple Algorithm for Local Register Allocation

- We execute the following for each three-address instruction in a basic block (in turn).

- Initially, the set `availReg` contains all usable physical registers.

```
# Allocate registers to an instruction x := y op z
# [Adopted from Aho, Sethi, Ullman]
regAlloc(x := y op z):
    if x has an assigned register already or dies here:
        return
    if y is a virtual register and dies here:
        reassign y's physical register to x
    elif availReg is not empty:
        remove a register from availReg and assign to x
    elif op requires a register:
        spill another virtual register (which could be y or z),
            and reassign its physical register to x
    else:
        just leave x in memory
```

Function Prologue and Epilogue for the ia32

- Consider a function that needs K bytes of local variables and other compiler temporary storage for expression evaluation.

- We'll consider the case where we keep a frame pointer.

- Overall, the code for a function, F, looks like this:

F:
```
        pushl %ebp              # Save dynamic link (caller's frame pointer)
        movl  %esp,%ebp         # Set new frame pointer
        subl  K,%esp            # Reserve space for locals
        code for body of function, leaving value in %eax
        leave                   # Sets %ebp to 0(%ebp), popping old frame pointer
        ret                     # Pop return address and return
```

Code Generation for Local Variables

- Local variables are stored on the stack (thus not at fixed location).

- One possibility: access relative to the stack pointer, but

  – Sometimes convenient for stack pointer to change during execution of of function, sometimes by unknown amounts.

  – Debuggers, unwinders, and stack tracers would like simple way to compute stack-frame boundaries.

- Solution: use frame pointer, which is constant over execution of function.

- For simple language, use fact that parameter i is at location frame pointer $+ K_1(i + K_2)$. If parameters are 32-bit integers (or pointers) on the ia32, $K_1 = 4$ and $K_2 = 2$ [why?].

- Local variables other than parameters are at negative offsets from the frame pointer on the ia32.

Accessing Non-Local Variables

- In program on left, how does f3 access x1?

- Let's suppose that functions pass static links as the first parameter of their callees.

- The static link passed to f3 will be f2's frame pointer.

```
def f1 (x1):
    def f2 (x2):
        def f3 (x3):
            ... x1 ...
        ...
        f3 (12)
    ...
    f2 (9)
```

# To access x1:
  movl 8(%ebp),%ebx # Fetch FP for f2
  movl 8(%ebx),%ebx # Fetch FP for f1
  movl 12(%ebx),%eax # Fetch x1
# When f2 calls f3:
  compute regular parameters
  pushl %ebp   # Pass f2's frame to f3
  call f3

- We'll say a function is at nesting level 0 if it is at the outer level, and at level $k + 1$ if it is most immediately enclosed inside a level-k function. Likewise, the variables, parameters, and code in a level-k function are themselves at level $k+1$ (enclosed in a level-k function).

- In general, for code at nesting level n to access a variable at nesting level $m \leq n$, perform $n - m$ loads of static links.

Accessing Non-Local Variables (II)

- The GNU convention for passing the static link is slightly different: it is passed in register `ecx`, making it easy to ignore if not needed. We'll use that in what follows.

```
def f1 (x1):
    def f2 (x2):
        def f3 (x3):
            ... x1 ...
        ...
        f3 (12)
    ...
    f2 (9)
```

Immediately after prologue:
  pushl %ecx   # Save static link at -4 off %ebp.

# To access x1:
  movl -4(%ebp),%ebx # Fetch FP for f2
  movl -4(%ebx),%ebx # Fetch FP for f1
  movl 8(%ebx),%eax   # Fetch x1

When f2 calls f3:
  compute parameters
  movl %ebp, %ecx   # Pass f2's frame to f3
  call f3

Calling Function-Valued Variables and Parameters

- As we've seen, a function value consists of a code address and a static link (let's assume code address comes first).

- So, in project 3, when we need the value of a function itself:

  ```
  def caller(f):
      f(42)
  ```

  we create an object containing the type pointer for the function type of `f`, and the code pointer and static link for `f`, and pass a pointer to this object.

- Then the call `f(42)` gets translated to

  ```
  pushl $42
  movl  8(%ebp), %eax    # Get parameter f
  movl  8(%eax), %ecx    # Fetch static link from f
  movl  4(%eax), %eax    # Get code address for f
  call  *%eax            # GNU assembler for call to address in eax
  ```

Static Links for Calling Known Functions

- For a call $F(\ldots)$ to a fixed, known function F, we could use the same strategy as for function-values variables:

  - Create a closure for F containing address of F's code and value of its static link.

  - Call F using the same code sequence as on previous slide.

- But can do better. Functions and their nesting levels are known.

- In code that is at nesting level n, to call a function at known nesting level $m \leq n$, get correct static link in register R with:

  - movl %ebp,R

  - Do 'movl -4(R),R' $n - m + 1$ times.

  (assuming we save static links at -4 off our frame pointer).

- When calling outer-level functions, it doesn't matter what you use as the static link.

Passing Static Links to Known Functions: Example

```
def f1 (x1):
   def f2 (x2):
      def f3 (x3):
         ... f2 (9) ...
      ...
      f3 (12)
      f2 (10) # (recursively)
   ...
```

```
# To call f2(9) (in f3):
  pushl $9
                       movl        -4(%ebp),%ebx  # Fetch FP for f2
                       movl        -4(%ebx),%ecx  # Fetch FP for f1, and pass it
  call  f2
  addl  $4,%esp


# To call f3(12) (in f2):
  pushl $12
  movll %ebp, %ecx    # f2's FP is static link
  call  f3
  addl  $4,%ebp


# To call f2(10) (in f2):
  pushl $10
                       movl        -4(%ebp),%ecx  # Pass down same static link
  call  f2
  addl  $4,%ebp
```

Variations

- Nothing forces us to pass static links this way.

- GCC on the ia32, for example, uses register `%ecx` to pass the static link (which the callee may then have to save).

- This makes the common case—calling an outer-level function—faster.

- In fact, for its own reasons, GCC passes an offset from the frame pointer as the static link (I'm not sure why). Clearly, it doesn't matter as long as the compiler is careful to use the static link consistently.

A Note on Pushing

- Don't really need to push and pop the stack as I've been doing.

- Instead, when allocating local variables, etc., on the stack, leave sufficient extra space on top of the stack to hold any parameter list in the function.

- Eg., to translate

  ```
  def f(x):
      g(g(x+2))
  ```

- We could either get the code on the left (pushing and popping) or that on the right (ignoring static links):

```
f:    movl  8(%ebp),%eax          f:    subl  $4,%esp
      addl  $2,%eax                      movl  8(%ebp),%eax
      pushl %eax                         addl  $2,%eax
      call  g                            movl  %eax,0(%esp)
      addl  $4,%esp                      call  g
      pushl %eax                         movl  %eax,0(%esp)
      call  g                            call  g
      addl  $4,%esp
```

...and you can continue to use the depressed stack pointer for arguments on the right.

Parameter Passing Semantics: Value vs. Reference

- So far, our examples have dealt only with *value parameters*, which are the only kind found in C, Java, and Python

  Ignorant comments from numerous textbook authors, bloggers, and slovenly hackers notwithstanding [End Rant].

- Pushing a parameter's value on the stack creates a copy that essentially acts as a local variable of the called function.

- C++ (and Pascal) have *reference parameters*, where assignments to the formal are assignments to the actual.

- Implementation of reference parameters is simple:

  - Push the address of the argument, not its value, and
  - To fetch from or store to the parameter, do an extra indirection.

Copy-in, Copy-out Parameters

- Some languages, such as Fortran and Ada, have a variation on this:
  copy-in, copy-out. Like call by value, but the final value of the param-
  eter is copied back to the original location of the actual parameter
  after function returns.

  - "Original location" because of cases like `f(A[k])`, where `k` might
    change during execution of `f`. In that case, we want the final
    value of the parameter copied back to `A[k0]`, where `k0` is the
    original value of `k` before the call.

  - Question: can you give an example where call by reference and
    copy-in, copy-out give different results?

Parameter Passing Semantics: Call by Name

- Algol 60's definition says that the effect of a call $P(E)$ is as if the body of P were substituted for the call (dynamically, so that recursion works) and E were substituted for the corresponding formal parameter in the body (changing names to avoid clashes).

- It's a simple description that, for simple cases, is just like call by reference:

```
procedure F(x)              F(aVar);
    integer x;          becomes
begin                       aVar := 42;
    x := 42;
end F;
```

- But the (unintended?) consequences were "interesting".

Call By Name: Jensen's Device

- Consider:

```
procedure DoIt (i, L, U, x, x0, E)
    integer i, L, U; real x, x0, E;
begin
    x := x0;
    for i := L step 1 until U do
        x := E;
end DoIt;
```

- To set y to the sum of the values in array A[1:N],

```
integer k;
DoIt(k, 1, N, y, 0.0, y+A[k]);
```

- To set z to the Nth harmonic number:

```
DoIt(k, 1, N, z, 0.0, z+1.0/k);
```

- Now how are we going to make this work?

Call By Name: Implementation

- Basic idea: Convert call-by-name parameters into parameterless functions (traditionally called *thunks*.)

- To allow assignment, these functions can return the addresses of their results.

- So the call

  ```
  DoIt(k, 1, N, y, 0.0, y+A[k]);
  ```

  becomes something like (please pardon highly illegal notation):

  ```
  integer t1;  real t2, t3, t4;
  t2 := 1.0; t3 := 0.0;
  DoIt(lambda: &k, lambda: &t2, lambda: &N, lambda: &y,
       lambda: &t3, lambda: (t4 := y+A[k], &t4));
  ```

- Later languages have abandoned this particular parameter-passing mode.