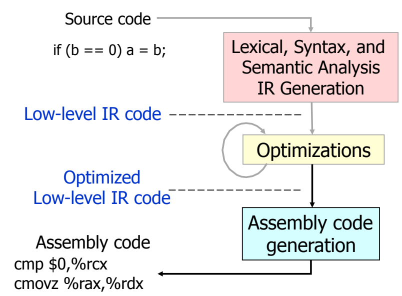
CS 434T: Code Generation

Stephen Freund
Williams College

1

Where We Are



2

Low IR to Assembly Translation

- Low IR code (TAC):
  - Variables (and temporaries)
  - No run-time stack
  - No calling sequences
  - Some abstract set of instructions

```
t3 = this.x
t3 = t2 * t3
t0 = t1 + t2
r = t0
t4 = w + 1
k = t4
```

- Translation
  - Calling sequences:
    - Translate function calls and returns
    - Manage run-time stack
  - Variables:
    - globals, locals, arguments, etc. assigned HW location
  - Instruction selection:
    - map sets of low level IR instructions to instructions in the target machine

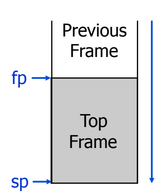
3

x86_64 Quick Overview

- Few registers:
  - 64 bits: `rax, rbx, rcx, rdx, rsi, rdi, r9-r15`
    - Also 16-bit: `eax, ebx, ecx, edx, esi, edi`
    - Also 16-bit: `ax, bx`, etc.
    - Also 8-bit: `al, ah, bl, bh`, etc.
  - Stack registers: `rsp, rbp`
- Many instructions:
  - Arithmetic: `add, sub, mod, idiv, imul`, etc.
  - Logic: `and, or, not, xor`
  - Comparison: `cmp, test`
  - Control flow: `jmp, jcc, jecz`
  - Function calls: `call, ret`
  - Data movement: `mov` (many variants)
  - Stack manipulations: `push, pop`
  - Other: `lea`
- Usually add "q" to indicate 64-bit: `addq, movq, cmpq`

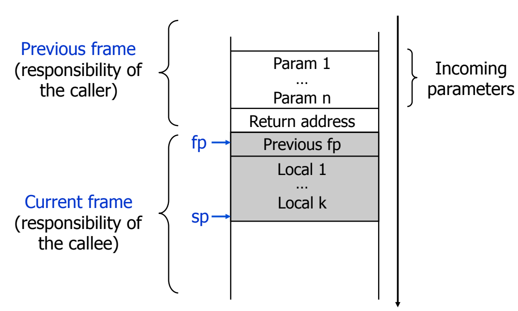
Stack Pointers

- Usually run-time stack grows downwards
  - Address of top of stack decreases
- Values in current frame accessed using two pointers:
  - Stack pointer (sp): points to frame top
  - Frame pointer(fp): points to frame base
  - Variable access: use offset from fp
- Why have both sp and fp?



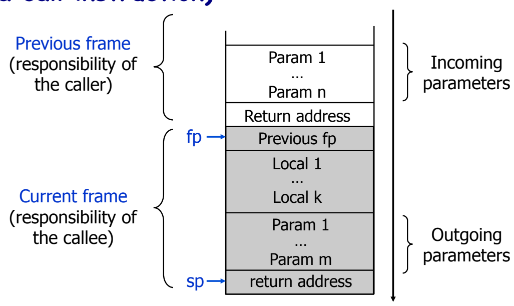
5

Anatomy of a Stack Frame



6

Anatomy of a Stack Frame (right after a call instruction)



7

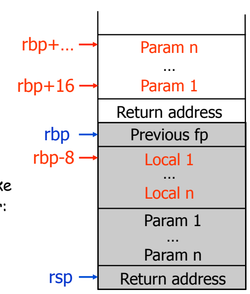
x86 Stack Management

- Stack pointer register: `%rsp`
- Frame pointer register: `%rbp`
- Push instructions: `push`, `pusha`, etc.
- Pop instructions: `pop`, `popa`, etc.
- Call instruction: `call`
- Return instruction: `ret`

8

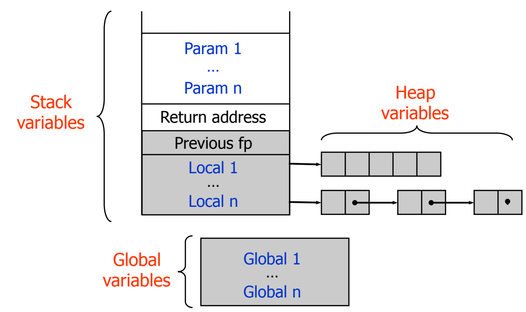
Accessing Stack Variables

- To access stack variables: use offsets from fp
- Example:
  `16(%rbp)` = parameter 1
  `24(%rbp)` = parameter 2
  `-8(%rbp)` =local 1
- Translate low-level code to take into account the frame pointer:
  `a = p+1`
  ⇒
  `-8(%rbp) = 32(%rbp)+1`



9

Big Picture: Memory Layout



10

Saving Registers During Function Calls

- Problem: execution of invoked function may overwrite useful values in registers
- Possibilities:
  - Callee saves and restores registers
  - Caller saves and restores registers
  - … or both

11

x86_64 ABI

- Defines calling conventions and register usage to ensure binary compatibility of compiled code
- Official Rules:
  - callee save: %rbx, %rbp, %r11-%r15
  - caller save: %rax, %rcx, %rdx, %edi, %esi, %r8-%r10
  - first six parameters to function passed in: %edi, %esi, %rdx, %rcx, %r8, %r9
- IC Rules:
  - callee/caller are the same
  - all parameters passed on stack
  - simpler, makes optimization / register allocation easier.

12

Example

- Consider call `foo(3, 5)`:
  - `%rcx` caller-saved
  - `%rbx` callee-saved
  - result passed back in `%rax`
- Code before call instruction:

```
push %rcx          // push caller saved registers
push $5            // push second parameter
push $3            // push first parameter
call _foo          // push ret. addr. & jump to callee
```

- Prologue at start of function:

```
push %rbp          // push old fp
mov %rsp, %rbp     // compute new fp
sub $24, %rsp      // push 3 integer local variables
push %rbx          // push callee saved registers
```

Only push those that are used after call

Only push those that are overwritten in function

13

Example

- Epilogue and end of function:

```
pop %rbx           // restore callee-saved registers
mov %rbp,%rsp      // pop callee frame, including locals
pop %rbp           // restore old fp
ret                // pop return address and jump
```

- Code after call instruction:

```
add $16,%rsp       // pop parameters
pop %rcx           // restore caller-saved registers
                   // rax contains return result
```

14

Simple Code Generation

- Three-address code makes it easy to generate assembly
  - (Not so easy to go directly from AST)

e.g. `a = p+q` ⇒

```
mov 16(%rbp), %rcx
add 8(%rbp), %rcx
mov %rcx, -8(%rbp)
```

- Need to consider many language constructs:
  - Operations: arithmetic, logic, comparisons
  - Accesses to local variables, global variables
  - Array accesses, field accesses
  - Control flow: conditional and unconditional jumps
  - Method calls, dynamic dispatch
  - Dynamic allocation (new)
  - Run-time checks

15

Arithmetic

- How to translate: `p+q` ?
  - Assume p and q are locals or parameters
  - Determine offsets for `p` and `q`
  - Perform the arithmetic operation
- Problem: the ADD instruction in x86 cannot take both operands from memory; notation for possible operands:
  - `add mem64, reg64`
  - `add reg64, mem64`
  - `add reg64, reg64`
  - `add imm64, reg64`
  - `...`
- Translation requires using an extra register
  - Place `p` into a register (e.g. `%rcx`): `mov 16(%rbp), %rcx`
  - Perform addition of `q` and `%rcx`: `add 8(%rbp), %rcx`

16

Data Movement

- Translate `a = p+q`:
  - Load memory location (`p`) into register (`%rcx`) using a move instr.
  - Perform the addition
  - Store result from register into memory location (`a`):

```
mov 16(%rbp), %rcx   (load)
add 8(%rbp), %rcx    (arithmetic)
mov %rcx, -8(%rbp)   (store)
```

- Move instructions:
  - cannot take both operands from memory

`a = p` ⇒

```
mov 16(%rbp), %rcx
mov %rcx, -8(%rbp)
```

- Loading constants:

`a = 12` ⇒ `mov $12, -8(%rbp)`

17

Control-Flow

- Label instructions
  - Simply translated as labels in the assembly code
  - E.g., `label2: mov $2, %rbx`
- Unconditional jumps:
  - Use jump instruction, with a label argument
  - E.g., `jmp label2`
- Conditional jumps:
  - Translate conditional jumps using `test/cmp` instructions:
  - E.g., `tjump b L` -> `cmp %rcx, $0`
    `jnz L`

    where `%rcx` holds the value of `b`, and we assume booleans are represented as 0=false, 1=true

18

Accessing Global Variables

- Global (static) variables are not allocated on the run-time stack
- Have fixed addresses throughout the execution of the program
  - Compile-time known addresses (relative to the base address where program is loaded)
  - Directly refer to addresses using symbolic names in the generated assembly code
- Example: string constants

```
mooStrData: .ascii "moo!"      # string data
mooStr:   .quad  mooStrData    # ptr to string data
```

  - The string will be allocated in the static area of the program
  - Can use `str` as a constant in other instructions:

```
movq mooStr(%rip), %rax
```

19

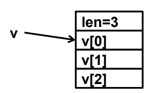
Accessing Heap Data

- Heap data allocated with new (Java) or malloc (C/C++)
  - Allocation function returns address of allocated heap data
  - Access heap data through that reference
- Array accesses in Java
  - access `a[i]` requires:
    - computing address of element: `a + i * size`
    - accessing memory at that address
  - Indexed memory accesses do it all
  - Example: assume size of array elements is 8 bytes, and local variables a, i (offsets –8, -16)

```
a[i] = 1    ⇒   mov -8(%rbp), %rbx       (load a)
                mov -16(%rbp), %rcx      (load i)
                mov $1, (%rbx,%rcx,8)    (store into the heap)
```

20

Run-time Checks



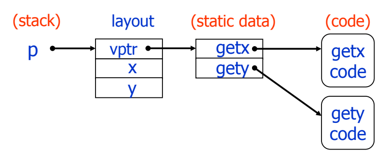
- Run-time checks:
  - Check if array/object references are non-null
  - Check if array index is within bounds
- Example: array bounds checks:
  - if v holds the address of an array, insert array bounds checking code for v before each load (…=v[i]) or store (v[i] = …)
  - Array length is stored just before array elements:

```
cmp $0, -24(%rbp)          (compare i to 0)
jl ArrayBoundsError        (test lower bound)
mov -16(%rbp), %rcx        (load v into %ecx)
mov -8(%rcx), %rcx         (load array length into %ecx)
cmp -24(%rbp), %rcx        (compare i to array length)
jle ArrayBoundsError       (test upper bound)
...
```

21

Object Layout

- Object consists of:
  - Methods
  - Fields
- Layout:
  - Pointer to VT, which contains pointers to methods
  - Fields.



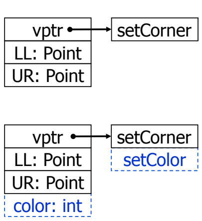
22

Field Offsets

- Offsets of fields from beginning of object known statically, same for all subclasses

```
class Shape {
  Point LL /* 8 */ , UR; /* 16 */
  void setCorner(Point p);
}

class ColoredRect extends Shape {
  Color c; /* 24 */
  void setColor(Color c);
}
```



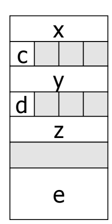
23

Field Alignment

- In many processors, a 32-bit load must be to an address divisible by 4, address of 64-bit load must be divisible by 8
- If permitted at all, unaligned accesses are usually much slower

⇒ Fields should be aligned

```
struct {
   int x;
   char c;
   int y;
   char d;
   int z; double e;
}
```



24

Dispatch Vector Lookup

C <: B <: A

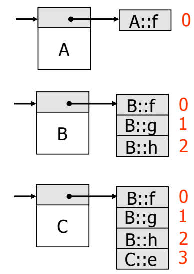
A	f
B	f,g,h
C	f,g,h,e

```
class A {
      void f() {...}    0
}
class B extends A {
      void f() {…}      0
      void g() {…}      1
      void h() {…}      2
}
class C extends B {
      void e() {…}      3
}
```

25

Dispatch Vector Layouts

- Index of f is the same in any object of type T <: A
- Methods may have multiple implementations
  - For subclasses with unrelated types
  - If subclass overrides method
- To execute a method m:
  - Lookup entry m in vector
  - Execute code pointed to by entry value



26

Code Generation: Virtual Tables

- Statically allocate one vtable per class

```
.data
ListVT: .quad _List_first
        .quad _List_rest
        .quad _List_length
```

27

Method Arguments

- Receiver object is (implicit) argument to method

```
class A {
  int f(int x,
        int y)
      { … }
}
```

compile as

```
int f(A this,
      int x,
      int y)
    { … }
```

28

Code Generation: Method Calls

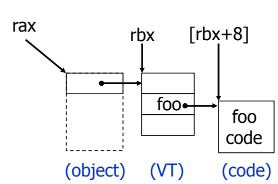
- Pre-function-call code:
  - Save registers
  - Push parameters
  - call function by its label
- Pre-method call:
  - Save registers
  - Push parameters
  - *Push receiver object reference*
  - *Lookup method in dispatch vector*

29

Example

```
o.foo(2,3);
```

```
push $3
push $2
push %rax
mov (%rax), %rbx
call *8(%rbx)
add $24, %rsp
```



compiler knows offset of foo in table

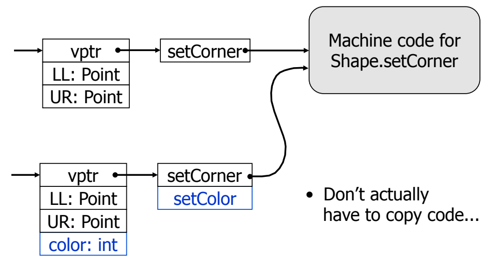
30

Interfaces, Abstract Classes

- Interfaces
  - no implementation
  - no dispatch vector info
  - (slow lookup a la SmallTalk)
- Abstract classes are halfway:
  - define some methods
  - leave others unimplemented
  - no objects (instances) of abstract class
  - Can construct vtable- just leave abstract entries "blank"

31

Code Sharing



- Don't actually have to copy code...

32

Code Generation: Library Calls

- Pass params in registers
  - %rdi for first param
  - %rsi for second param
- Return result is in %rax
- Warning: library functions may modify caller save registers

```
movq $100, %rdi
call __LIB_printi

…

movq $20, %rdi
call __LIB_random
movq %rax, -32(%rbp)
```

33

Code Generation: Allocation

- Heap allocation: o = new C()
  - Allocate heap space for object
  - Store pointer to vtable into newly allocated memory

```
movq $32, %rdi # 3 fields+vptr
call __LIB_allocObject
leaq _C_VT(%rip), %rdi
movq %rdi, (%rax)
```

34

35