Winter 2006-2007 Compiler Construction T10 – IR part 3 + Activation records

Mooly Sagiv and Roman Manevich
School of Computer Science
Tel-Aviv University

Today

ic
IC Language
Lexical Analysis
Syntax Analysis Parsing
AST
Symbol Table etc.
Inter. Rep. (IR)
Code Generation
exe
Executable code

- Today:
  - LIR
    - Spec. + microLIR
    - Strings and arrays
    - PA4
    - Optimizations
  - Runtime organization
    - Activation records
- Next time:
  - Introduction to x86 assembly
  - Code generation
  - Activation records in depth

LIR language

- Supports
  - Literal strings
  - Dispatch tables
  - Instruction set
  - Unbounded number of registers
  - Object/array allocation via library functions
    - Updating DVPtr with LIR instructions
    - Missing from printout: `ArrayLength` instruction
    - Notice special syntax for parameter passing for static/virtual function calls
- No representation of activation records (frames)
- No calling sequence protocols
  - Just Call/Return instructions
  - `this` variable "magically" updated on call to virtual functions

Translating call/return

```
TR[C.foo(e1,…,en)]   R1 := TR[e1]
                     …
                     Rn := TR[en]
                     StaticCall C_foo(x1=R1,…,xn=Rn),R

TR[e1.foo(e2)]       R1 := TR[e1]
                     R2 := TR[e2]
                     VirtualCall R1.Cfoo(x=R2),R

TR[return e]         R1 := TR[e]
                     Return R1
```

- formal parameter name (points to x1)
- actual argument register (points to R1)
- Constant representing offset of method f in dispatch table of class type of e1 (points to c_{foo})

4

LIR translation example

```
class A {
  int x;
  string s;
  int foo(int y) {
    int z=y+1;
    return z;
  }
  static void main(string[] args) {
    A p = new B();
    p.foo(5);
  }
}

class B extends A {
  int z;
  int foo(int y) {
    s = "y=" + Library.itos(y);
    Library.println(s);
    int[] sarr = Library.stoa(s);
    int l = sarr.length;
    Library.printi(l);
    return l;
  }
}
```

- Translating virtual functions (dispatch tables)
- Translating the main function
- Translating virtual function calls
- Translation for literal strings
- Translating `.length` operator

5

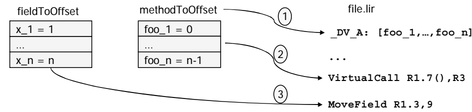
LIR program (manual trans.)

```
str1: "y="                            # Literal string in program
_DV_A: [_A_foo]                       # dispatch table for class A
_DV_B: [_B_foo]                       # dispatch table for class B

_A_foo:                               # int foo(int y)
Move y,R1                             # int z=y+1;
Add 1,R1
Move R1,z
Return z                              # return z;

_B_foo:                               # int foo(int y)
Library __itos(y),R1                  # Library.itos(y);
Library __stringCat(str1,R1),R2       # "y=" + Library.itos(y);
Move this,R3                          # this.s = "y=" + Library.itos(y);
MoveField R2,R3.3
MoveField R3.3,R4
Library __println(R4),Rdummy
Library __stoa(R4),R5                 # int[] sarr = Library.stoa(s);
Move R5,sarr
ArrayLength sarr,R6                   # int l = sarr.length;
Move R6,l
Library __printi(l),Rdummy            # Library.printi(l)
Return l                              # return l;

# main in A
_ic_main:                             # A {static void main(string[] args)…}
Library __allocateObject(16),R1       # A p = new B();
MoveField _DV_B,R1.0                  # Update DVPtr of new object
VirtualCall R1.0(y=5),Rdummy          # p.foo(5)
```

6

Class layout implementation

```
class A {
  int x_1;
  ...
  boolean x_n;
  void foo_1(…) {…}
  ...
  int foo_n(…) {…}
}
```



1: generate dispatch tables
2: determine method offsets in virtual calls
3: determine field offsets in field access statements

fieldToOffset
x_1 = 1
…
x_n = n

methodToOffset
foo_1 = 0
…
foo_n = n-1

file.lir

```
_DV_A: [foo_1,…,foo_n]        (1)
...
VirtualCall R1.7(),R3         (2)
MoveField R1.3,9              (3)
```

7

microLIR simulator

- Java application
  - Accepts file.lir (your translation)
  - Executes program
- Use it to test your translation
  - Checks correct syntax
  - Performs lightweight semantic checks
  - Runtime semantic checks
  - Debug modes (-verbose:1/2)
  - Prints program statistics (#registers)
- Comes with sample inputs
- Read manual
- Comes with sources (allowed to use in PA4)
- Not heavily tested (better than nothing)

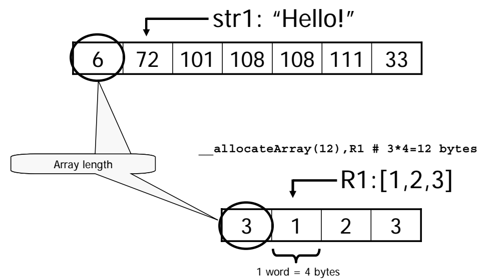
8

PA4

- Translate AST to LIR (file.ic -> file.lir)
  - Dispatch table for each class
  - Literal strings (all literal strings in file.ic)
  - Instruction list for every function
    - Leading label for each function _CLASS_FUNC
  - Label of main function should be `_ic_main`
- Maintain internally for each function
  - List of LIR instructions
  - Reference to method AST node
    - Needed to generate frame information in PA5
- Maintain for each call instruction
  - Reference to method AST
    - Needed to generate call sequence in PA5
- Optimizations (WARNING: only after assignment works)
  - Keep optimized and non-optimized translations separately

9

Representing arrays/strings



10

LIR optimizations

- Aim to reduce number of LIR registers
  - Reduce size of activation records
  - Allow better register allocation in PA5
  - Also reduces number of instructions
- Avoid storing variables and constants in registers
- Use accumulator registers
- Reuse "dead" registers
- Weighted register allocation
- (More complicated: reuse assigned values in block)
- Merge consecutive labels
  - Left to PA5 (generates additional labels)

11

Avoid storing constants and variables in registers

- Naïve translation of AST leaves
  - For a constant TR[5] = Move 5,Rj
  - For a variable TR[x] = Move x,Rk
- Better translation
  - For a constant TR[5] = 5
  - For a variable TR[x] = x
  - What about TR[x+5]=?
    - **WRONG:** TR[x+5] = Add TR[x],TR[5] = Add x,5
    - TR[x+5] = Move 5,R1
Add x,R1
    - Assign to register if both operands non-registers

12

Accumulator registers

- Use same register for sub-expression and result
- Very natural for 2-address code and previous optimization

```
TR[e1 OP e2]
```

Naive translation

```
R1 := TR[e1]
R2 := TR[e2]
R3 := R1 OP R2
```

Better translation

```
R1 := TR[e1]
R2 := TR[e2]
R1 := R1 OP R2
```

13

Accumulator registers

```
TR[e1 OP e2]
a+(b*c)
```

Naive translation

```
R1 := TR[e1]
R2 := TR[e2]
R3 := R1 OP R2
```

```
Move a,R1
Move b,R2
Mul R1,R2
Move R2,R3
Move c,R4
Add R3,R4
Move R4,R5
```

Better translation

```
R1 := TR[e1]
R2 := TR[e2]
R1 := R1 OP R2
```

```
Move b,R1
Mul c,R1
Add a,R1
```

14

Accumulator registers cont.

- Accumulating instructions, use:
  - **MoveArray R1[R2],R1**
  - **MoveField R1.7,R1**
  - **StaticCall _foo(R1,…),R1**
  - …

15

Reuse registers

- Registers have very-limited lifetime
  - Currently stored values used exactly once
  - (All LIR registers become dead after statement)
- Suppose TR[e1 OP e2] translated as
  R1:=TR[e1], R2:=TR[e2], R1:=R1 OP R2
  Registers from TR[e1] can be reused in TR[e2]
- Algorithm:
  - Use a stack of temporaries (LIR registers)
  - Stack corresponds to recursive invocations of t := TR[e]
  - All the temporaries on the stack are alive

16

Reuse registers cont.

- Implementation: use counter c to implement live register stack
  - Registers R(0)…R(c) are alive
  - Registers R(c+1),R(c+2)… can be reused
  - Push means increment c, pop means decrement c
- In the translation of R(c)=TR[e1 OP e2]

```
R(c)  := TR[e1]
                -------------- c = c + 1
R(c)  := TR[e2]
                -------------- c = c - 1
R(c)  := R(c) OP  R(c+1)
```

17

Example

```
        R0 := TR[((c*d)-(e*f))+(a*b)]

               ------c = 0

                              R0 := c*d
                                            ------c = c + 1
R0 := TR[(c*d)-(e*f)]         R1 := e*f
                                            ------c = c - 1
                              R0 := R0-R1

               ------c = c + 1
R1 := a*b
               ------c = c - 1
R0 := R0 + R1
```

18

Weighted register allocation

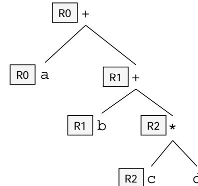
- Suppose we have expression e1 OP e2
  - e1,e2 without side-effects (function calls, assignments)
  - OP is commutative: *,+
  - TR[e1 OP e2] = TR[e2 OP e1]
  - Does order matter?
- Use the Sethi & Ullman algorithm
  - Weighted register allocation – translate heavier sub-tree first

19

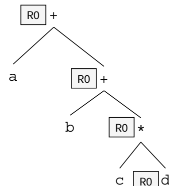
Example

```
R0 := TR[a+(b+(c*d))]
```

left child first



right child first



Translation uses all optimizations shown until now uses 3 registers

Managed to save two registers register

20

Weighted register allocation

- Can save registers by re-ordering (commutative) subtree computations
- Label each node with its weight
  - Weight = number of registers needed
  - Leaf weight known
  - Internal node weight
    - w(left) > w(right) then w = left
    - w(right) > w(left) then w = right
    - w(right) = w(left) then w = left + 1
- Choose heavier child as first to be translated
- WARNING: have to check that no side-effects exist before attempting to apply this optimization (pre-pass on the tree)

21

LIR vs. assembly

	LIR	Assembly
#Registers	Unlimited	Limited
Function calls	Implicit	Runtime stack
Instruction set	Abstract	Concrete
Types	Basic and user defined	Limited basic types

Actually very limited in our LIR

22

Function calls

- LIR – simply call/return
- Conceptually
  - Supply new environment (frame) with temporary memory for local variables
  - Pass parameters to new environment
  - Transfer flow of control (call/return)
  - Return information from new environment (ret. value)
- Assembly – pass parameters (+this), save registers,call, restore registers, return, pass return value, virtual method lookup

23

Activation records

- New environment = activation record (a.k.a. frame)
- Activation record = data of current function / method call
  - User data
    - Local variables
    - Parameters
    - Return values
    - Register contents
  - Administration data
    - Code addresses
    - Pointers to other activation records (not in IC)
- In IC – a stack is sufficient !

24

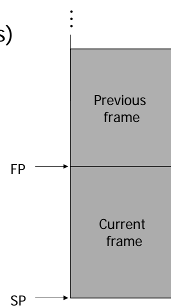
Runtime stack

- Stack of activation records
- Call = push new activation record
- Return = pop activation record
- Only one "active" activation record – top of stack
- This is enough to handle recursion

25

Runtime stack

- Stack grows downwards (towards smaller addresses)
- SP – stack pointer – top of current frame
- FP – frame pointer – base of current frame
  - Sometimes called BP (base pointer)



26

Pentium runtime stack

Register	Usage
ESP	Stack pointer
EBP	Base pointer

Pentium stack registers

Instruction	Usage
push, pusha,…	Push on runtime stack
pop, popa,…	Pop from runtime stack
call	Transfer control to called routine
ret	Transfer control back to caller

Pentium stack and call/ret instructions

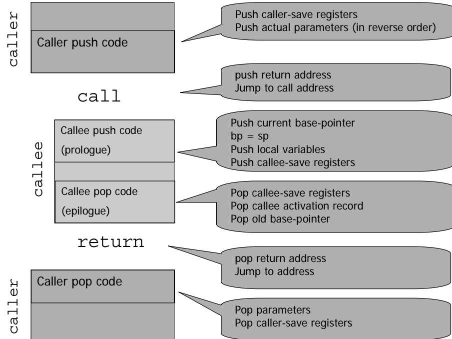
27

Call sequences

- The processor does not save the content of registers on procedure calls
- So who will?
  - Caller saves and restores registers
  - Callee saves and restores registers
  - But can also have both save/restore some registers

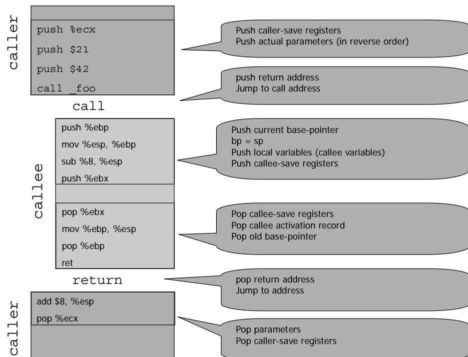
28

Call sequences



29

Call sequences – `Foo(42,21)`



30

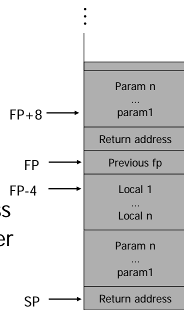
"To Callee-save or to Caller-save?"

- Callee-saved registers need only be saved when callee modifies their value
- Some conventions exist (cdecl)
  - %eax, %ecx, %edx – caller save
  - %ebx, %esi, %edi – callee save
  - %esp – stack pointer
  - %ebp – frame pointer
  - Use %eax for return value

31

Accessing stack variables

- Use offset from EBP
- Remember – stack grows downwards
- Above EBP = parameters
- Below EBP = locals
- Examples
  - %ebp + 4 = return address
  - %ebp + 8 = first parameter
  - %ebp – 4 = first local



32

Happy new year!

33