

# Compiler Construction

## Intermediate Representation III Activation Records

Ran Shaham and Ohad Shacham  
School of Computer Science  
Tel-Aviv University

---

---

---

---

---

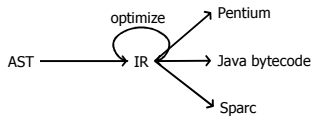
---

---

---

## Intermediate representation

- Allows language-independent, machine independent optimizations and transformations
- Easy to translate from AST
- Easy to translate to assembly



2

---

---

---

---

---

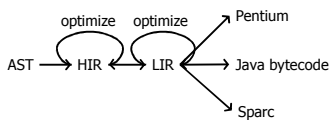
---

---

---

## Multiple IRs

- Some optimizations require high-level structure
- Others more appropriate on low-level code
- Solution: use multiple IR stages



3

---

---

---

---

---

---

---

---

## LIR optimizations

- Aim to reduce number of LIR registers and number of instructions
- Avoid storing variables and constants in registers
- Use accumulator registers
- Reuse "dead" registers
- Weighted register allocation

4

---

---

---

---

---

---

---

---

## Avoid storing constants and variables in registers

- Don't allocate target register for each instruction
  - $TR[5] = \text{Move } 5, R_j$
  - $TR[x] = \text{Move } x, R_k$
- For a constant  $TR[5] = 5$
- For a variable  $TR[x] = x$
- $TR[x+5] = \text{Move } 5, R_1$   
Add  $x, R_1$ 
  - Assign to register if **both operands** non-registers

5

---

---

---

---

---

---

---

---

## Accumulator registers

- Use same register for sub-expression and result

$TR[e1 \text{ OP } e2]$

$R1 := TR[e1]$

$R2 := TR[e2]$

$R1 := R1 \text{ OP } R2$

6

---

---

---

---

---

---

---

---

## Accumulator registers

TR[e1 OP e2]

a+(b\*c)

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

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

7

---

---

---

---

---

---

---

---

## Accumulator registers cont.

- For instruction with N registers dedicate one register for accumulation
- Accumulating instructions, use:
  - `MoveArray R1[R2], R1`
  - `MoveField R1.7, R1`
  - `StaticCall _foo(R1, ...), R1`
  - ...

8

---

---

---

---

---

---

---

---

## Reuse registers

- Registers have very-limited lifetime
- TR[e1 OP e2] =  
R1:=TR[e1]  
R2:=TR[e2]  
R1:=R1 OP R2  
  
Registers from TR[e1] can be reused in TR[e2]
- Solution:
  - Use a stack of LIR registers
  - Stack corresponds to recursive invocations of t := TR[e]
  - All the temporaries on the stack are alive

9

---

---

---

---

---

---

---

---

## Weighted register allocation

- [Sethi & Ullman algorithm](#)
  - Two expression e1, e2 and an operation OP
  - e1,e2 without side-effects
    - function calls
  - $TR[e1 \text{ OP } e2] = TR[e2 \text{ OP } e1]$
- Weighted register allocation
  - translate heavier sub-tree first

10

---

---

---

---

---

---

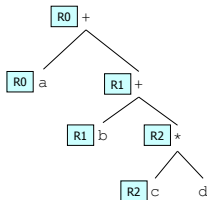
---

---

## Example

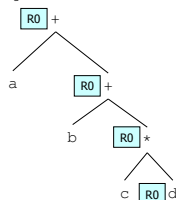
$R0 := TR[a+(b+(c*d))]$

left child first



Translation uses all optimizations shown until now uses 3 registers

right child first



Managed to save two registers

11

---

---

---

---

---

---

---

---

## Weighted register allocation

- Can save registers by re-ordering subtree computations
- Label each node with its weight
  - Weight = number of registers needed
  - Leaf weight known
  - Internal node weight
    - $w(\text{left}) > w(\text{right})$  then  $w = \text{left}$
    - $w(\text{right}) > w(\text{left})$  then  $w = \text{right}$
    - $w(\text{right}) = w(\text{left})$  then  $w = \text{left} + 1$
- Choose heavier child as first to be translated
- Have to check that no side-effects exist

12

---

---

---

---

---

---

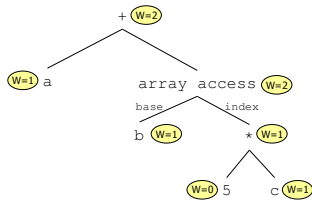
---

---

## Weighted reg. alloc. example

$R0 := TR[a+b[5*c]]$

Phase 1: - check absence of side-effects in expression tree  
- assign weight to each AST node



13

---

---

---

---

---

---

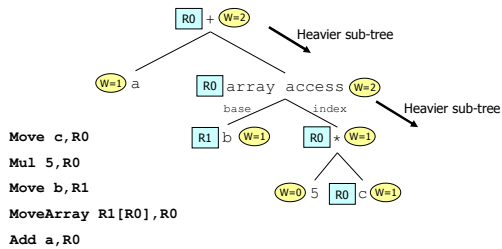
---

---

## Weighted reg. alloc. example

$R0 := TR[a+b[5*c]]$

Phase 2: use weights to decide on order of translation



14

---

---

---

---

---

---

---

---

## 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

15

---

---

---

---

---

---

---

---

## Tips for PA4

- Keep **list** of LIR instructions for each translated method
- Keep **ClassLayout** information for each class
  - Field offsets
  - Method offsets
  - Don't forget to take superclass fields and methods into account
- Two AST passes:
  - Pass 1:
    - Collect and name strings literals (`Map<ASTStringLiteral, String>`)
    - Create `ClassLayout` for each class
  - Pass 2: use literals and field/method offsets to translate method bodies
- Finally: print string literals, dispatch tables, print translation list of each method body

16

---

---

---

---

---

---

---

---

## microLIR simulator

- Written by Roman Manevich
- 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, #labels, etc.`)
- Comes with sample inputs
- Read manual

17

---

---

---

---

---

---

---

---

## LIR vs. assembly

	LIR	Assembly
#Registers	Unlimited	Limited
Function calls	Implicit	Runtime stack
Instruction set	Abstract	Concrete

18

---

---

---

---

---

---

---

---

## Function calls

- 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)

19

---

---

---

---

---

---

---

---

## 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

20

---

---

---

---

---

---

---

---

## Runtime stack

- Stack of activation records
- Call = push new activation record
- Return = pop activation record
- Only one “active” activation record – top of stack

21

---

---

---

---

---

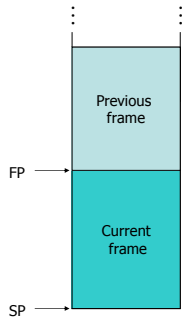
---

---

---

## 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)



22

---

---

---

---

---

---

---

---

## X86 runtime stack

Register	Usage
ESP	Stack pointer
EBP	Base pointer

X86 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

X86 stack and call/ret instructions

23

---

---

---

---

---

---

---

---

## Call sequences

- The processor does not save the content of registers on procedure calls
- So who will?
  - Caller saves and restores registers
    - Caller's responsibility
  - Callee saves and restores registers
    - Callee's responsibility

24

---

---

---

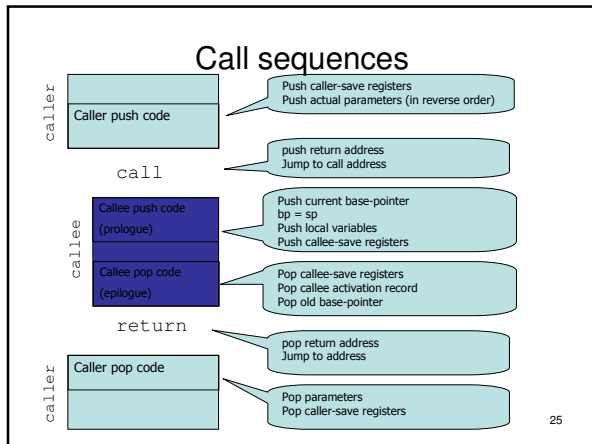
---

---

---

---

---




---

---

---

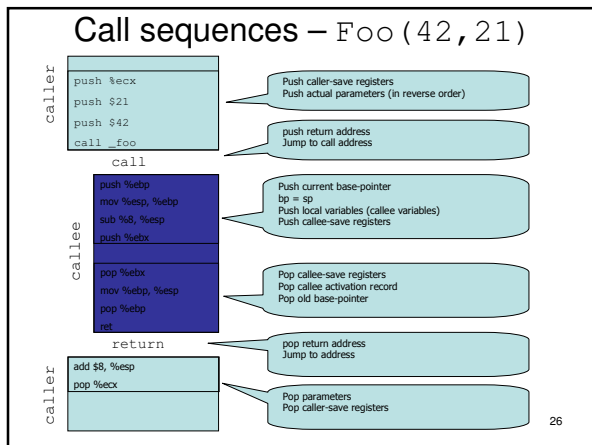
---

---

---

---

---




---

---

---

---

---

---

---

---

- ### “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
- 27

---

---

---

---

---

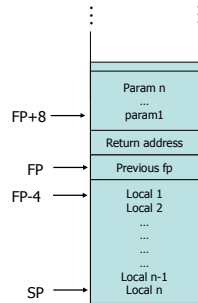
---

---

---

## Accessing stack variables

- Use offset from EBP
- Stack grows downwards
- Above EBP = parameters
- Below EBP = locals



- Examples
  - `%ebp + 4` = return address
  - `%ebp + 8` = first parameter
  - `%ebp - 4` = first local

28

---

---

---

---

---

---

---

---

---

---

## main calling method bar

```
int bar(int x)
{
    int y;
    ...
}
static void main(string[] args) {
    int z;
    Foo a = new Foo();
    z = a.bar(31);
    ...
}
```

29

---

---

---

---

---

---

---

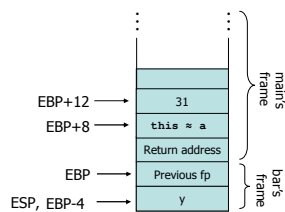
---

---

---

## main calling method bar

```
int bar(Foo this, int x)
{
    int y;
    ...
}
static void main(string[] args) {
    int z;
    Foo a = new Foo();
    z = a.bar(a, 31);
    ...
}
```



- Examples
  - `%ebp + 4` = return address
  - `%ebp + 8` = first parameter
    - Always `this` in virtual function calls
  - `%ebp` = old `%ebp` (pushed by callee)
  - `%ebp - 4` = first local

30

---

---

---

---

---

---

---

---

---

---