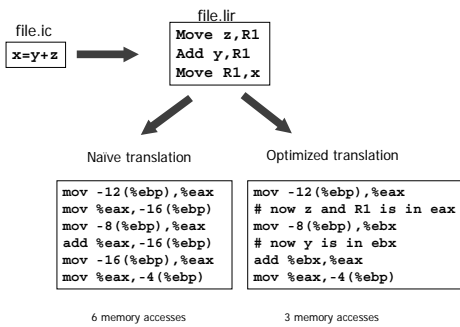


## A simple register allocation optimization scheme

## Register allocation example



## Optimizing register allocation

- Goal: associate machine registers with LIR registers as much as possible
- Optimization done per IC statement (sequence of LIR instructions translated from same statement)
- Basic idea: store first 6 LIR registers in machine registers and the rest in frame
  - But decide on which ones during translation
- Track association between LIR registers and machine registers using RegMap
- Use values stored in RegMap to translate LIR instructions
- Flush register values back to frame if needed

## RegMap

- Main data structure is RegMap – a map from x86 registers to Free/Reg:
  - Free – x86 register is currently available
  - Reg – LIR register
  - U – used by some variable
- Operations supported by RegMap:
  - RegMap.get : x86Reg -> LIR register or Free
  - RegMap.get : LIR register -> x86Reg or null
  - RegMap.put(x86Reg,LIRReg) – create an association
  - RegMap.getFreeReg – returns an available x86 registers if there is any, otherwise flush a register back to frame and return an available register
    - E.g., if edx is associated with R3 with offset -12 then  
emit mov %edx,-12(%ebp)  
and return edx

4

---

---

---

---

---

---

---

---

## Definitions

- For each LIR instruction, define:
  - Read registers – LIR registers storing values used by the instruction
  - Write registers – LIR registers storing result of instruction
- Example: **Add op1, op2**
  - Read registers = {op1,op2} ∩ LIRRegs
  - Write registers = {op2} ∩ LIRRegs
  - op1 can be LIRReg/Immediate/Memory
  - op2 can be LIRReg/Memory

5

---

---

---

---

---

---

---

---

## Translating Add op1, op2

```
GetOp(op) : immediate or x86 register
if op is Immediate
  return immediate
else if op is Memory with offset op_offset
  xreg = RegMap.getFreeReg()
  emit mov op_offset(%ebp),xreg
  return xreg
else if op is LIRReg with offset op_offset
  if (xreg,op) in RegMap
    return xreg
  else
    xreg = RegMap.getFreeReg()
    emit mov op_offset(%ebp),xreg
    RegMap.put(xreg,op)
```

```
Translate Add op1,op2
val1 = GetOp(op1)
val2 = GetOp(op2)
emit add val1,val2
if op2 is Memory
  emit mov val2,op2_offset(%ebp)
```

6

---

---

---

---

---

---

---

---

## Translation examples

AX	BX	CX	DX	SI	DI
F	F	F	F	F	F

Move z, R1

```
mov 8(%ebp), %eax
```

AX	BX	CX	DX	SI	DI
R1	F	F	F	F	F

Move z, R7

AX	BX	CX	DX	SI	DI
R1	R2	R3	R4	R5	R6

```
mov %eax, -12(%ebp) #R1
mov 8(%ebp), %eax
```

Store R1 back  
in frame

AX	BX	CX	DX	SI	DI
R7	R2	R3	R4	R5	R6

7

---

---

---

---

---

---

---

---

## Translation examples

AX	BX	CX	DX	SI	DI
R1	R2	R3	R4	R5	R6

Move y, R1

```
mov 12(%ebp), %eax
```

AX	BX	CX	DX	SI	DI
R1	R2	R3	R4	R5	R6

Add R2, R1

```
add %ebx, %eax
```

AX	BX	CX	DX	SI	DI
R1	R2	R3	R4	R5	R6

is this entry going  
to be used again?

8

---

---

---

---

---

---

---

---

## Translation algorithm

- Identify sequences of LIR instructions that correspond to same IC statement
- Initialize RegMap to be empty
- Translate each LIR instruction using RegMap
- If LIR register Rx is read and not write can clear its entry
  - Special cases: LIR registers used for conditions in if/while statements – used in **Compare** instructions
- When flushing register to frame how do we choose which one?
  - Decide on policy, e.g., least frequently used

9

---

---

---

---

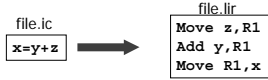
---

---

---

---

## Example revisited



Optimized translation

```
# Move z,R1      map={}
mov -12(%ebp),%eax
#               map={eax=R1}
# Add y,R1
mov -8(%ebp),%ebx # use free register ebx for y
#               map={eax=R1,ebx=U}
add %ebx,%eax
#               map={eax=R1}
# Move R1,x
mov %eax,-4(%ebp)
#               R1 read and not write -
#               clear entry
#               map={}
```

10

---

---

---

---

---

---

---

---

## Conclusion

- Very naive scheme for register allocation during code generation
  - Works better if number of LIR registers small (PA4 optimizations)
  - Can generalize to basic blocks – sequence of LIR instructions without labels and jumps
  - Really just a hack
- Better technique – liveness analysis with graph coloring
  - Minimizes number of registers for entire function (not just single statement)
  - Can allocate same register for different local variables

11

---

---

---

---

---

---

---

---

**That's all folks**

12

---

---

---

---

---

---

---

---