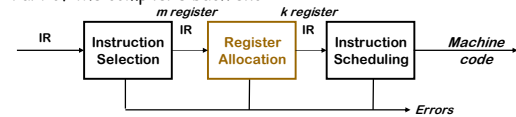


Register Allocation

Register Allocation

Part of the compiler's back end



Critical properties

- Produce **correct** code that uses k (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold **spilled values**
- Operate efficiently
 $O(n)$, $O(n \log_2 n)$, maybe $O(n^2)$, but not $O(2^n)$

CS430

2

Register Allocation

- Motivation
 - Registers much faster than memory
 - Limited number of physical registers
 - Keep values in registers as long as possible
 - Minimize number of load / store statements executed
- Register allocation & assignment
 - For simplicity
 - Assume infinite number of virtual registers
 - Decide which values to keep in finite # of virtual registers
 - Assign virtual registers to physical registers

CS430

3

Register Allocation

The Task

- At each point in the code, pick the values to keep in registers
- Insert code to move values between registers & memory
 - No transformations (leave that to scheduling)
- Minimize inserted code
 - Use both dynamic & static measures
- Make good use of any **extra** registers

Allocation versus assignment

- **Allocation** is deciding which values to keep in registers
- **Assignment** is choosing specific registers for values
- This distinction is often lost in the literature

The compiler must perform both allocation & assignment

CS430

4

Register Allocation Approaches

- Local allocation (within basic blocks)
 - Top-down
 - Assign registers by frequency
 - Bottom-up
 - Spill registers by reuse distance
- Global allocation (across basic blocks)
 - Top-down
 - Color interference graph
 - Bottom-up
 - Split live ranges

CS430

5

Local Register Allocation

- What's "local"? (as opposed to "global")
 - A local transformation operates on basic blocks
 - Many optimizations are done locally
- Does local allocation solve the problem?
 - It produces decent register use inside a block
 - Inefficiencies can arise at boundaries between blocks
- How many passes can the allocator make?
 - This is an off-line problem
 - As many passes as it takes
- Memory-to-memory vs. register-to-register model
 - Code shape and safety issues

CS430

6

Register Allocation

Can we do this optimally? (on real code?)

Local Allocation

- Simplified cases $\Rightarrow O(n)$
- Real cases $\Rightarrow$ NP-Complete

Local Assignment

- Single size, no spilling $\Rightarrow O(n)$
- Two sizes $\Rightarrow$ NP-Complete

Global Allocation

- NP-Complete for 1 register
- NP-Complete for k registers
(most sub-problems are NPC, too)

Global Assignment

- NP-Complete

Real compilers face real problems

CS430

7

Observations

Allocator may need to reserve registers to ensure feasibility

- Must be able to compute addresses
- Requires some minimal (feasible) set of registers, F
 $\rightarrow F$ depends on target architecture
- Use these registers only for spilling
(set them "aside", i.e., not available for register assignment)

Notation:

k is the number of registers on the target machine

What if $k - F < |\text{values}| < k$?

- The allocator can either
 $\rightarrow$ Check for this situation
 $\rightarrow$ Accept the fact that the technique is an approximation

CS430

8

Observations

A value is **live** between its **definition** and its **uses**

- Find definitions ($x \leftarrow \dots$) and uses ($y \leftarrow \dots x \dots$)
- From definition to **last** use is its **live range**
 $\rightarrow$ How does a second definition affect this?
- Can represent live range as an interval $[i, j]$ (in block)
 $\rightarrow$ live on exit

Let $MAXLIVE$ be the maximum, over each instruction i in the block, of the number of values (pseudo-registers) live at i .

- If $MAXLIVE \leq k$, allocation should be easy
- If $MAXLIVE \leq k$, no need to reserve F registers for spilling
- If $MAXLIVE > k$, some values must be spilled to memory
Finding live ranges is harder in the global case

CS430

9

ILOC Instruction Set

Operation	Meaning	Latency
load $r1 \Rightarrow r2$	$MEM(r1) \rightarrow r2$	2
store $r1 \Rightarrow r2$	$r1 \rightarrow MEM(r2)$	2
loadl $c \Rightarrow r1$	$c \rightarrow r1$	1
add $r1, r2 \Rightarrow r3$	$r1 + r2 \rightarrow r3$	1
sub $r1, r2 \Rightarrow r3$	$r1 - r2 \rightarrow r3$	1
mult $r1, r2 \Rightarrow r3$	$r1 \times r2 \rightarrow r3$	1
lshift $r1, r2 \Rightarrow r3$	$r1 \ll r2 \rightarrow r3$	1
rshift $r1, r2 \Rightarrow r3$	$r1 \gg r2 \rightarrow r3$	1
output c	print out $MEM(c)$	1

Assume a register-to-register memory model, with 1 class of registers. Latencies are important for instruction scheduling, not register allocation and assignment

CS430

10

ILOC Example

Sample code sequence

```
loadI 1028 => r1 // r1 ← 1028
load r1 => r2 // r2 ← MEM(r1) == y
mult r1, r2 => r3 // r3 ← 1028 · y
loadI 5 => r4 // r4 ← 5
sub r4, r2 => r5 // r5 ← 5 - y
loadI 8 => r6 // r6 ← 8
mult r5, r6 => r7 // r7 ← 8 · (5 - y)
sub r7, r3 => r8 // r8 ← 8 · (5 - y) - (1028 · y)
store r8 => r1 // MEM(r1) ← 8 · (5 - y) - (1028 · y)
```

CS430

11

ILOC Example - Live Ranges

Live range for r1

```
1 loadI 1028 => r1 // r1
2 load r1 => r2 // r1 r2
3 mult r1, r2 => r3 // r1 r2 r3
4 loadI 5 => r4 // r1 r2 r3 r4
5 sub r4, r2 => r5 // r1 r3 r5
6 loadI 8 => r6 // r1 r3 r5 r6
7 mult r5, r6 => r7 // r1 r3 r7
8 sub r7, r3 => r8 // r1 r8
9 store r8 => r1 //
```

NOTE: live sets on **exit** of each instruction

CS430

12

ILOC Example - Live Ranges

➤ Live range for r2

```

1 loadI 1028 => r1 // r1
2 load   r1  => r2 // r1 r2
3 mult   r1, r2 => r3 // r1 r2 r3
4 loadI  5   => r4 // r1 r2 r3 r4
5 sub    r4, r2 => r5 // r1 r3 r5
6 loadI  8   => r6 // r1 r3 r5 r6
7 mult   r5, r6 => r7 // r1 r3 r7
8 sub    r7, r3 => r8 // r1 r8
9 store  r8    => r1 //

```

NOTE: live sets on exit of each instruction

CS430

13

ILOC Example - Live Ranges

➤ Live range for r3

```

1 loadI 1028 => r1 // r1
2 load   r1  => r2 // r1 r2
3 mult   r1, r2 => r3 // r1 r2 r3
4 loadI  5   => r4 // r1 r2 r3 r4
5 sub    r4, r2 => r5 // r1 r3 r5
6 loadI  8   => r6 // r1 r3 r5 r6
7 mult   r5, r6 => r7 // r1 r3 r7
8 sub    r7, r3 => r8 // r1 r8
9 store  r8    => r1 //

```

NOTE: live sets on exit of each instruction

CS430

14

Top-down Versus Bottom-up Allocation

Top-down allocator

- Work from external notion of what is important
- Assign registers in priority order
- Register assignment remains fixed for entire basic block
- Save some registers for the values relegated to memory (feasible set F)

Bottom-up allocator

- Work from detailed knowledge about problem instance
- Incorporate knowledge of partial solution at each step
- Register assignment may change across basic block
- Save some registers for the values relegated to memory (feasible set F)

CS430

15

Bottom-up Allocator

The idea:

- Focus on replacement rather than allocation
- Keep values "used soon" in registers

Algorithm:

- Start with empty register set
- Load on demand
- When no register is available, free one

Replacement:

- **Spill** the value whose next use is **farthest in the future**
- Prefer clean value to dirty value
- Sound familiar? Think cache line / page replacement ...

CS430

16

Spill code

- A virtual register is **spilled** by using only registers from the feasible set (F), not the allocated set (k-F)
- How to insert spill code, with $F = \{f1, f2, \dots\}$?
 - For the **definition** of the spilled value (assignment of the value to the virtual register), use a feasible register as the target register and then use an additional register to load its address in memory, and perform the store:


```
add r1, r2 => r3    add r1, r2 => f1
                    loadI @f => f2 // value lives at memory location @f
                    store f1 => f2
```
 - For the **use** of the spilled value, load value from memory into a feasible register:


```
add r1, r2 => r3    loadI @f => f1 // value lives at memory location @f
                    load f1 => f1
                    add f1, r2 => r3
```

- How many feasible registers do we need for an add instruction?

CS430, 2

17

ILOC Example - Bottom-up Allocation

➤ Bottom up (3 physical registers: ra, rb, rc)

source code			life ranges			register allocation and assignment(on exit)		
						ra	rb	rc
1	loadI	1028	=> r1	// r1		r1		
2	load	r1	=> r2	// r1 r2		r1	r2	
3	mult	r1, r2	=> r3	// r1 r2 r3		r1	r2	r3
4	loadI	5	=> r4	// r1 r2 r3 r4		r4	r2	r3
5	sub	r4, r2	=> r5	// r1 r5 r3		r4	r5	r3
6	loadI	8	=> r6	// r1 r5 r3 r6		r6	r5	r3
7	mult	r5, r6	=> r7	// r1 r7 r3		r6	r7	r3
8	sub	r7, r3	=> r8	// r1 r8		r6	r8	r3
9	store	r8	=> r1	//		r1	r8	r3

Part of r1 live range spilled

Note: this is only one possible allocation and assignment!

CS430

18

ILOC Example - Bottom-up Assignment

- Bottom up (3 physical registers: ra, rb, rc)

source code			register allocation and assignment(on exit)		
			ra	rb	rc
1	loadI	1028 => r1	r1		
2	load	r1 => r2	r1	r2	
3	mult	r1, r2 => r3	r1	r2	r3
4	loadI	5 => r4	r4	r2	r3
5	sub	r4, r2 => r5	r4	r5	r3
6	loadI	8 => r6	r6	r5	r3
7	mult	r5, r6 => r7	r6	r7	r3
8	sub	r7, r3 => r8	r6	r8	r3
9	store	r8 => r1	r1	r8	r3

Let's generate code now!

CS430

19

ILOC Example - Bottom-up Assignment

- Bottom up (3 physical registers: ra, rb, rc)

source code			register allocation and assignment(on exit)		
			ra	rb	rc
1	loadI	1028 => r3			r3
2	load	r1 => r2	r1	r2	
3	mult	r1, r2 => r3	r1	r2	r3
4	loadI	5 => r4	r4	r2	r3
5	sub	r4, r2 => r5	r4	r5	r3
6	loadI	8 => r6	r6	r5	r3
7	mult	r5, r6 => r7	r6	r7	r3
8	sub	r7, r3 => r8	r6	r8	r3
9	store	r8 => r1	r1	r8	r3

For written registers, use current register assignment

CS430

20

ILOC Example - Bottom-up Assignment

- Bottom up (3 physical registers: ra, rb, rc)

source code			register allocation and assignment(on exit)		
			ra	rb	rc
1	loadI	1028 => ra			
2	load	ra => r2	r1	r2	
3	mult	r1, r2 => r3	r1	r2	r3
4	loadI	5 => r4	r4	r2	r3
5	sub	r4, r2 => r5	r4	r5	r3
6	loadI	8 => r6	r6	r5	r3
7	mult	r5, r6 => r7	r6	r7	r3
8	sub	r7, r3 => r8	r6	r8	r3
9	store	r8 => r1	r1	r8	r3

For read registers, use previous register assignment

CS430

21

ILOC Example - Bottom-up Assignment

- Bottom up (3 physical registers: ra, rb, rc)

source code			register allocation and assignment(on exit)		
			ra	rb	rc
1	loadI	1028 => ra			
2	load	ra => r2	r1	r2	
3	mult	r1, r2 => r3	r1	r2	r3
4	loadI	5 => r4	r4	r2	r3
5	sub	r4, r2 => r5	r4	r5	r3
6	loadI	8 => r6	r6	r5	r3
7	mult	r5, r6 => r7	r6	r7	r3
8	sub	r7, r3 => r8	r6	r8	r3
9	store	r8 => r1	r1	r8	r3

CS430

22

ILOC Example - Bottom-up Assignment

- Bottom up (3 physical registers: ra, rb, rc)

source code			register allocation and assignment(on exit)		
			ra	rb	rc
1	loadI	1028 => ra	r1		
2	load	ra => rb	r1	r2	
3	mult	ra, rb => rc	r1	r2	r3
4	store*	ra => 10	r1	r2	r3
5	loadI	5 => r4	r4	r2	r3
6	sub	r4, r2 => r5	r4	r5	r3
7	loadI	8 => r6	r6	r5	r3
8	mult	r5, r6 => r7	r6	r7	r3
9	sub	r7, r3 => r8	r6	r8	r3
9	store	r8 => r1	r1	r8	r3

Insert spill code

CS430

23

ILOC Example - Bottom-up Assignment

- Bottom up (3 physical registers: ra, rb, rc)

source code			register allocation and assignment(on exit)		
			ra	rb	rc
1	loadI	1028 => ra			
2	load	ra => rb	r1	r2	
3	mult	ra, rb => rc	r1	r2	r3
4	store	ra => 10	r1	r2	r3
5	loadI	5 => ra	r4	r2	r3
6	sub	ra, rb => rb	r4	r5	r3
7	loadI	8 => ra	r6	r5	r3
8	mult	rb, ra => rb	r6	r7	r3
9	sub	rb, rc => rb	r6	r8	r3
9	store	r8 => r1	r1	r8	r3

CS430

24

ILOC Example - Bottom-up Assignment

➤ Bottom up (3 physical registers: ra, rb, rc)

source code				register allocation and assignment(on exit)		
				ra	rb	rc
1	loadI	1028	⇒ ra	r1		
2	load	ra	⇒ rb	r1	r2	
3	mult	ra, rb	⇒ rc	r1	r2	r3
3	store	ra	⇒ 10	r1	r2	r3
4	loadI	5	⇒ ra	r4	r2	r3
5	sub	ra, rb	⇒ rb	r4	r5	r3
6	loadI	8	⇒ ra	r6	r5	r3
7	mult	rb, ra	⇒ rb	r6	r7	r3
8	sub	rb, rc	⇒ rb	r6	r8	r3
9	load*	10	⇒ ra	r1	r8	r3
9	store	r8	⇒ r1	r1	r8	r3

Insert spill code

CS430

25

ILOC Example - Bottom-up Assignment

➤ Bottom up (3 physical registers: ra, rb, rc)

source code				register allocation and assignment(on exit)		
				ra	rb	rc
1	loadI	1028	⇒ ra	r1		
2	load	ra	⇒ rb	r1	r2	
3	mult	ra, rb	⇒ rc	r1	r2	r3
3	store*	ra	⇒ 10	r1	r2	r3
4	loadI	5	⇒ ra	r4	r2	r3
5	sub	ra, rb	⇒ rb	r4	r5	r3
6	loadI	8	⇒ ra	r6	r5	r3
7	mult	rb, ra	⇒ rb	r6	r7	r3
8	sub	rb, rc	⇒ rb	r6	r8	r3
9	load*	10	⇒ ra	r1	r8	r3
9	store	rb	⇒ ra	r1	r8	r3

Done

CS430

26

Top-down Allocator

The idea:

- Keep busiest values in a register
- Use the feasible (reserved) set, F , for the rest

Algorithm:

- Rank values by number of occurrences
- Allocate first $k - F$ values to registers
- Rewrite code to reflect these choices

SPILL: Move values with no register into memory (add LOADs & STOREs)

CS430

27

ILOC Example - Top-down Allocation

➤ Top down (3 physical registers: ra, rb, rc)

1	loadI	1028	⇒ r1	// r1		
2	load	r1	⇒ r2	// r1 r2		
3	mult	r1, r2	⇒ r3	// r1 r2 r3		
4	loadI	5	⇒ r4	// r1 r2 r3 r4		
5	sub	r4, r2	⇒ r5	// r1 r3 r5		
6	loadI	8	⇒ r6	// r1 r3 r5 r6		
7	mult	r5, r6	⇒ r7	// r1 r3 r7		
8	sub	r7, r3	⇒ r8	// r1 r7 r8		
9	store	r8	⇒ r1	//		

Note that this assumes that an extra register is not needed for save/restore

➤ Consider

-# of occurrences of virtual register (most important)

-Fewer → better spill candidate

-r1=3, r2=2, r3=2, r4=2, r5=2, r6=2, r7=2, r8=2

-Length of live range (tie breaker)

-Longer → better spill candidate

-r1=8, r2=3, r3=5, r4=2, r5=2, r6=1, r7=1, r8=1

CS430

28

ILOC Example - Top-down Assignment

➤ Top down (3 physical registers: ra, rb, rc)

1	loadI	1028	⇒ ra	// r1		
2	load	ra	⇒ rb	// r1 r2		
3	mult	ra, rb	⇒ rc	// r1 r2 r3		
3	store*	rc	⇒ 10	// spill code		
4	loadI	5	⇒ rc	// r1 r2 r3 r4		
5	sub	rc, rb	⇒ rb	// r1 r3 r5		
6	loadI	8	⇒ rc	// r1 r3 r5 r6		
7	mult	rb, rc	⇒ rb	// r1 r3 r7		
8	load*	10	⇒ rc	// spill code		
8	sub	rb, rc	⇒ rb	// r1 r8		
9	store	rb	⇒ ra	//		

Note that this assumes that an extra register is not needed for save/restore

➤ Insert spill code for every occurrence of spilled virtual register in basic block

CS430

29

Register Allocation Approaches

• Local allocation (within basic blocks)

→ Top-down

• Assign registers by frequency

→ Bottom-up

• Spill registers by reuse distance

• Global allocation (across basic blocks)

→ Top-down

• Color interference graph

→ Bottom-up

• Split live ranges

CS430

30

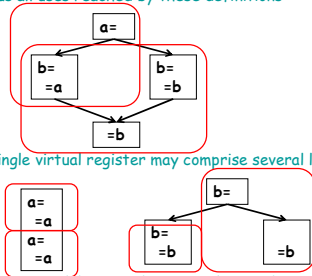
Global Register Allocation - Top Down

- Register coloring
 - Maps register allocation to graph coloring
 - Major steps
 1. Global data-flow analysis to find live ranges
 2. Build and color interference graph
 3. If unable to find coloring, spill registers & repeat

CS430

31

Global Live Ranges

- Definition
 - All definitions which reach a use...
 - ...plus all uses reached by these definitions
- 
- A single virtual register may comprise several live ranges
 - Live ranges delineate when variables need to be stored in the same physical register to avoid extra code

CS430

32

Interference

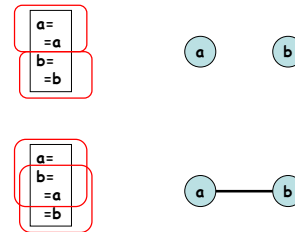
- Using live ranges, an **interference graph** is constructed where
 - Vertices represent live ranges
 - Edges represent interferences between live ranges
 - Both ranges are live at same point
 - Cannot occupy the same register
 - Coloring represents register assignment
 - One color per register
- Using a graph coloring abstraction subtly changes the problem
 - Justified by need to separate optimization and allocation

CS430

33

Building the Interference Graph

- Algorithm
 - At each point p in the program
 - Add edge (x,y) for all pairs of live ranges x, y live at p
- Example



CS430

34

Coloring

- Graph coloring
 - Given graph, find assignment of colors to each node
 - Such that no neighbors have the same color
 - Determining whether a graph has a k -coloring
 - Is NP-hard for $k > 2$
- Register coloring
 - Find a legal coloring given k colors
 - Where k is the number of available registers

CS430

35

Graph Coloring Through Simplification

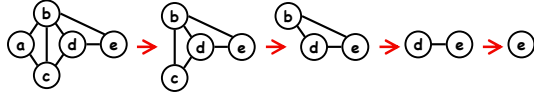
- Coloring algorithm [Chaitin et al., 1981]
 1. Repeatedly remove nodes with degree $< k$ from graph
 - Push nodes onto stack
 2. If every remaining node is degree $\geq k$
 - Spill node with lowest spill cost
 - Remove node from graph
 3. Reassemble graph with nodes popped from stack
 - As each node is added to graph
 - Choose a color differing from its neighbors

CS430

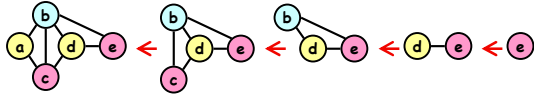
36

Graph Coloring Example

- Given interference graph and 3 registers
- Simplify graph by removing nodes with < 3 neighbors
→ Push nodes onto stack



- Reassemble graph by popping nodes from stack
→ Assigning colors not used by neighbors

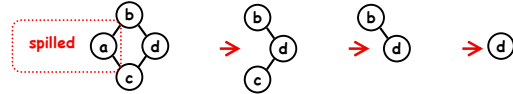


CS430

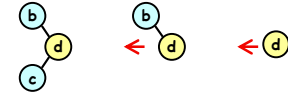
37

Graph Coloring

- Given interference graph and 2 registers
- Simplify graph by removing nodes with < 2 neighbors
→ No such node, must spill node with lowest spill cost
- Remaining nodes can then be simplified & colored



- Can we do better?
→ Yes!



CS430

38

Optimistic Graph Coloring

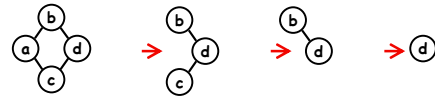
- Optimistic coloring algorithm [Briggs et al., 1989]
 - Remove nodes with degree $< k$ from graph (pop onto stack)
 - If every node has degree $> k$
 - Remove node with lowest spill cost (pop onto stack)
 - Reassemble graph with nodes popped from stack
 - Spill node if it cannot be colored
- Optimistic coloring defers spilling decision
 - Helps if neighbors of node
 - Are the same color
 - Have already been spilled

CS430

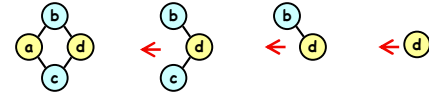
39

Optimistic Graph Coloring

- Given interference graph and 2 registers
- Simplify graph by removing nodes with < 2 neighbors
→ No such node, remove node with lowest spill cost
→ Continue graph simplification



- Reassemble graph by popping nodes from stack
→ Assigning colors not used by neighbors



CS430

40

Spill Code

- Inserted when too few registers to hold all live ranges
 - Insert load before use
 - Insert store after definition
- Effects
 - Breaks live range into many small live ranges
 - Reduces chance of interference
 - Expensive
 - Introduces load / store instructions
 - For each use / def instruction in live range

CS430

41

Spill Cost

- Need to decide which live range to spill if needed
- Two metrics to consider
 - Cost of spill
 - Cost of load / store instructions inserted
 - Decrease in interference
 - Reduce need for more spills

CS430

42

Spill Cost

- Possible cost functions

$degree(v)$	# of edges for v in interference graph
$depth(I)$	loop nesting depth of instruction I
$cost(v)$	$\sum_{v \text{ live at instr } I} 10^{depth(I)}$

Assumes
10 loop
iterations

- Possible cost estimate heuristics

→ Cost / degree
→ Cost / (degree * degree)
→ Etc... [Chaitin et al., 1981]

CS430

43

Allocation with Spilling

- One approach

→ Apply different cost estimate heuristics
▪ Pick best result
→ Assumptions
▪ Building interference graph is highest expense
▪ Spill cost estimates can be calculated inexpensively

- Reducing spill code by recognizing special cases

→ Value modified (dirty)
▪ Store register value to memory, reload for use
→ Read-only value (clean)
▪ Reload from memory for use
→ Constant value (rematerializable)
▪ Recompute value (no need for memory load!)

CS430

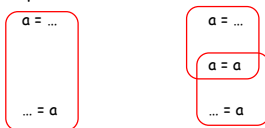
44

Global Register Allocation - Bottom Up

- Live range splitting

→ Insert copies to split up live ranges
▪ Hopefully reduces need for spilling
→ Also controls spill code placement
▪ Spill code generated at copies

- Examples



CS430

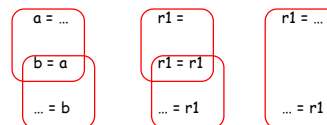
45

Global Register Allocation - Bottom Up

- Coalescing (subsumption)

→ Allocate source and destination of copy to same register
▪ To eliminate register-to-register copies
→ Combines live ranges
→ Can reverse unnecessary splits

- Examples



CS430

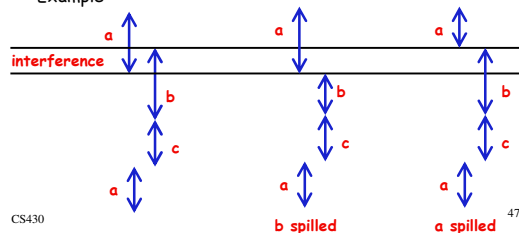
46

Live Range Splitting

- Approach [Chow & Hennessy 1990]

1. Locally allocate registers for each basic block
2. Prioritize live ranges by estimated spill cost
3. Allocate registers to live ranges
4. Split live range if no colors available

- Example

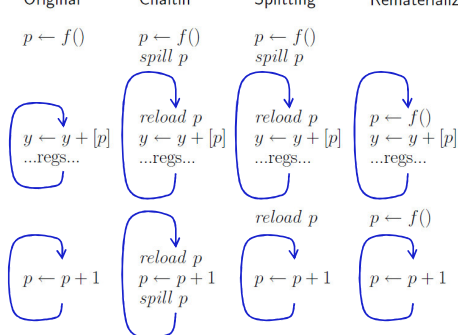


CS430

47

Register Allocation Examples

Original Chaitin Splitting Rematerialized



Assuming
only 1
register
available,
must spill p

CS430

48

Combining Instruction Scheduling & Register Allocation

- Allocation before scheduling
 - Register assignment introduces dependences
 - Anti & output dependences
 - Reduces freedom of instruction scheduling

- Example

load vr1,a	load r1,a	load r1,a
vr3 = vr1	r3 = r1	load r2,b
load vr2,b	load r1,b	r3 = r1
vr4 = vr2	r4 = r1	r4 = r2

CS430

49

Combining Instruction Scheduling & Register Allocation

- Scheduling before allocation
 - Lengthens live range of virtual registers
 - Increases register pressure
 - May cause spills
 - Still need to schedule spill code after allocation

- Example

load vr1,a	load vr1,a
vr4 = vr1	load vr2,b
load vr2,b	load vr3,c
vr5 = vr2	vr4=vr1
load vr3,c	vr5=vr2
vr6 = vr3	vr6=vr3

CS430

50

Scheduling and Allocation Are Interdependent

- Conflicting goals for scheduling & allocation
- Some possible solutions
 - Assigning registers
 - First fit
 - Lowest available register number
 - Reduces total number of registers used
 - Round robin
 - Cycle through all registers
 - Reduces memory-related dependences

CS430

51

Scheduling and Allocation Are Interdependent

- More possible solutions
 - Change ordering
 - Postpass - allocate then schedule
 - Prepass - schedule then allocate
 - Multipass - schedule, allocate, schedule
 - Integrated prepass scheduling
 - Schedule instructions first as preparation
 - Bias schedule to reduce local register pressure
 - Allocate registers after scheduling

CS430

52