Instruction Scheduling: Beyond Basic Blocks

COMP 412
Fall 2005

Copyright 2005, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Local Scheduling

As long as we stay within a single block

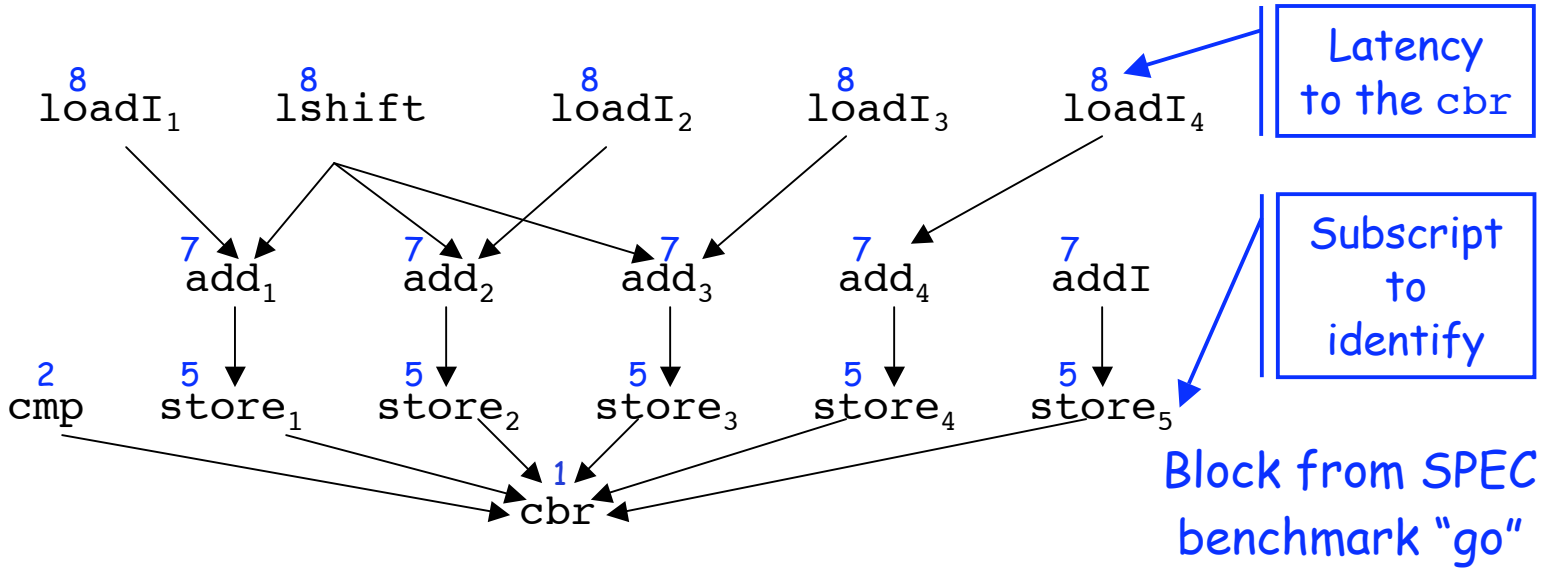
- List scheduling does well

- Problem is hard, so tie-breaking matters
  - More descendants in dependence graph
  - Prefer operation with a last use over one with none
  - Breadth first makes progress on all paths
    - Tends toward more ILP & fewer interlocks
  - Depth first tries to complete uses of a value
    - Tends to use fewer registers

Classic work on this is Gibbons & Muchnick ([154] in EaC)

Local Scheduling

Forward and backward can produce different results



Operation	load	loadI	add	addI	store	cmp
Latency	1	1	2	1	4	1

Local Scheduling

Forward Schedule

	Int	Int	Mem
1	$loadI_1$	lshift	
2	$loadI_2$	$loadI_3$	
3	$loadI_4$	add_1	
4	add_2	add_3	
5	add_4	addI	$store_1$
6	cmp		$store_2$
7			$store_3$
8			$store_4$
9			$store_5$
10			
11			
12			
13	cbr		

Backward Schedule

	Int	Int	Mem
1	$loadI_4$		
2	addI	lshift	
3	add_4	$loadI_3$	
4	add_3	$loadI_2$	$store_5$
5	add_2	$loadI_1$	$store_4$
6	add_1		$store_3$
7			$store_2$
8			$store_1$
9			
10			
11	cmp		
12	cbr		
13			

Using latency to root as the priority

Local Scheduling

Schielke's RBF algorithm

- Run 5 passes of forward list scheduling and 5 passes of backward list scheduling
- Break each tie randomly
- Keep the best schedule
  - Shortest time to completion
  - Other metrics are possible (*shortest time + fewest registers*)

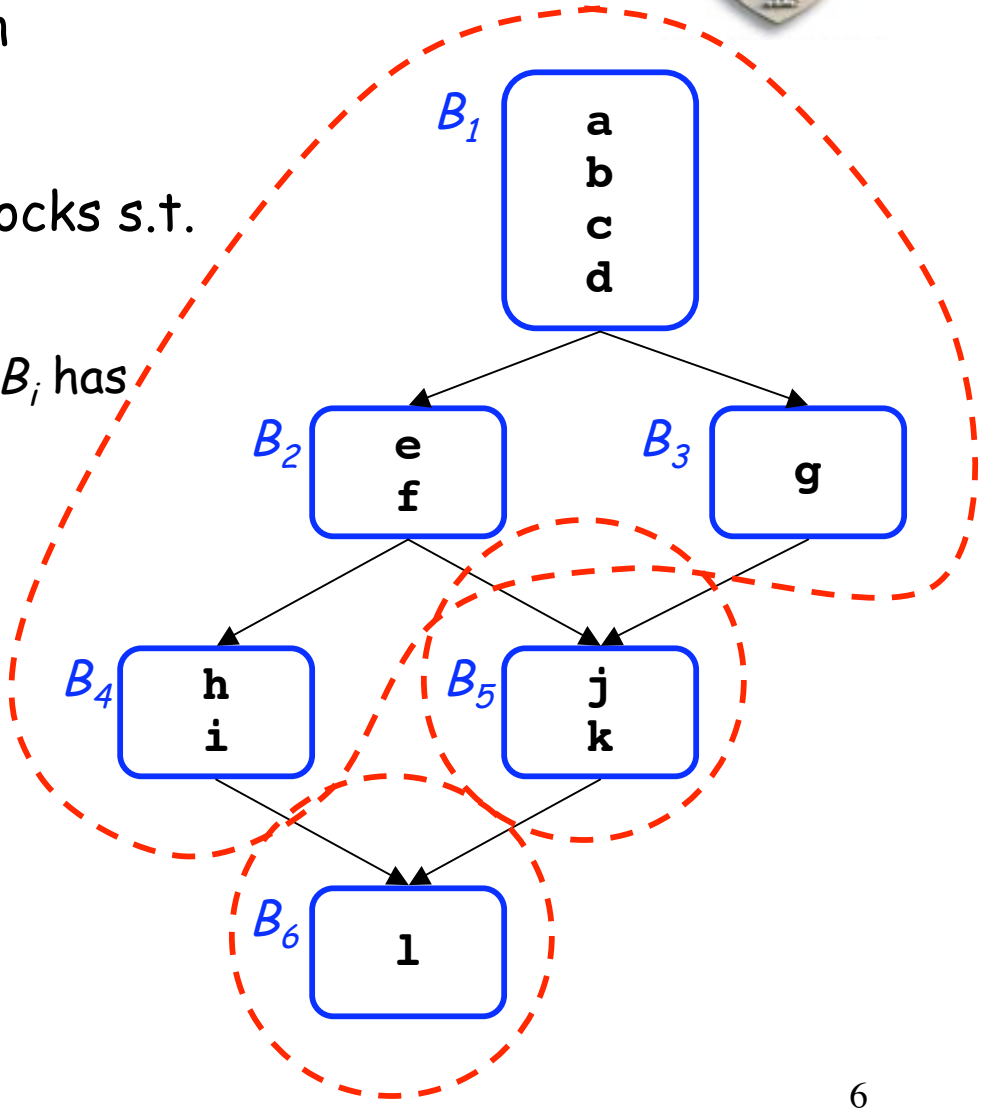
Randomized Backward & Forward

In practice, this does very well

Scheduling Larger Regions

One step beyond a block is an Extended Basic Block (EBB)

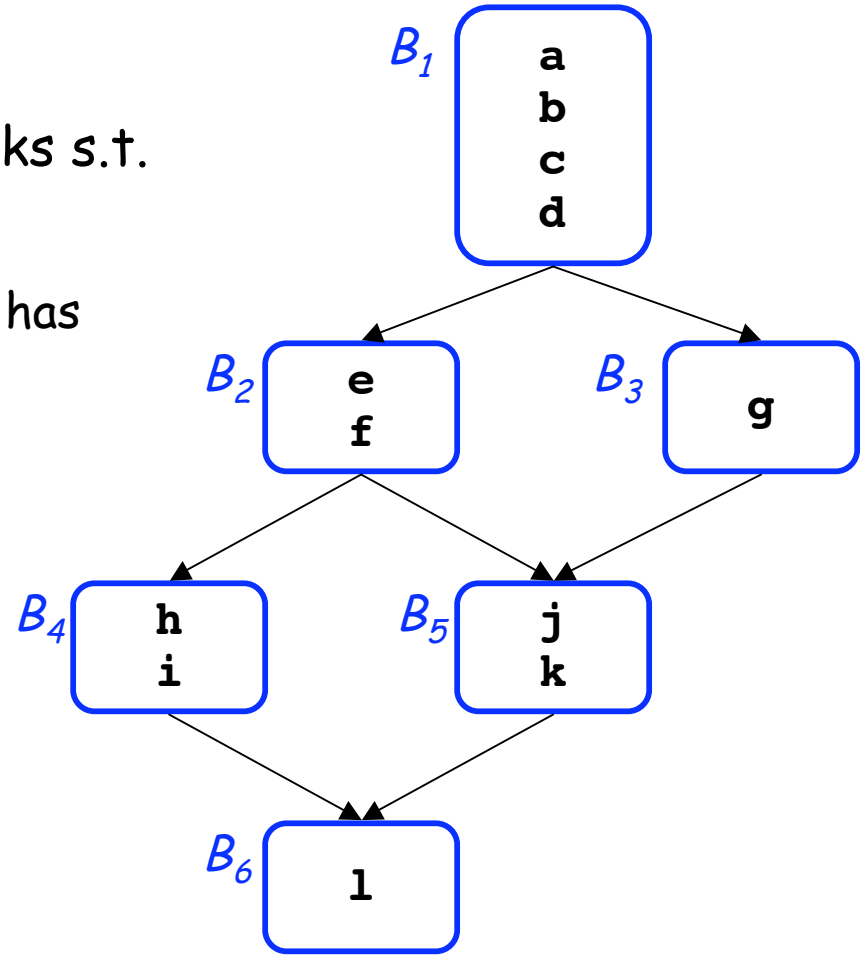
- EBB is a maximal set of blocks s.t.
  - Set has a single entry, B_i
  - Each block B_j other than B_i has exactly one predecessor
- Example has three EBBs



Scheduling Larger Regions

One step beyond a block is an Extended Basic Block (EBB)

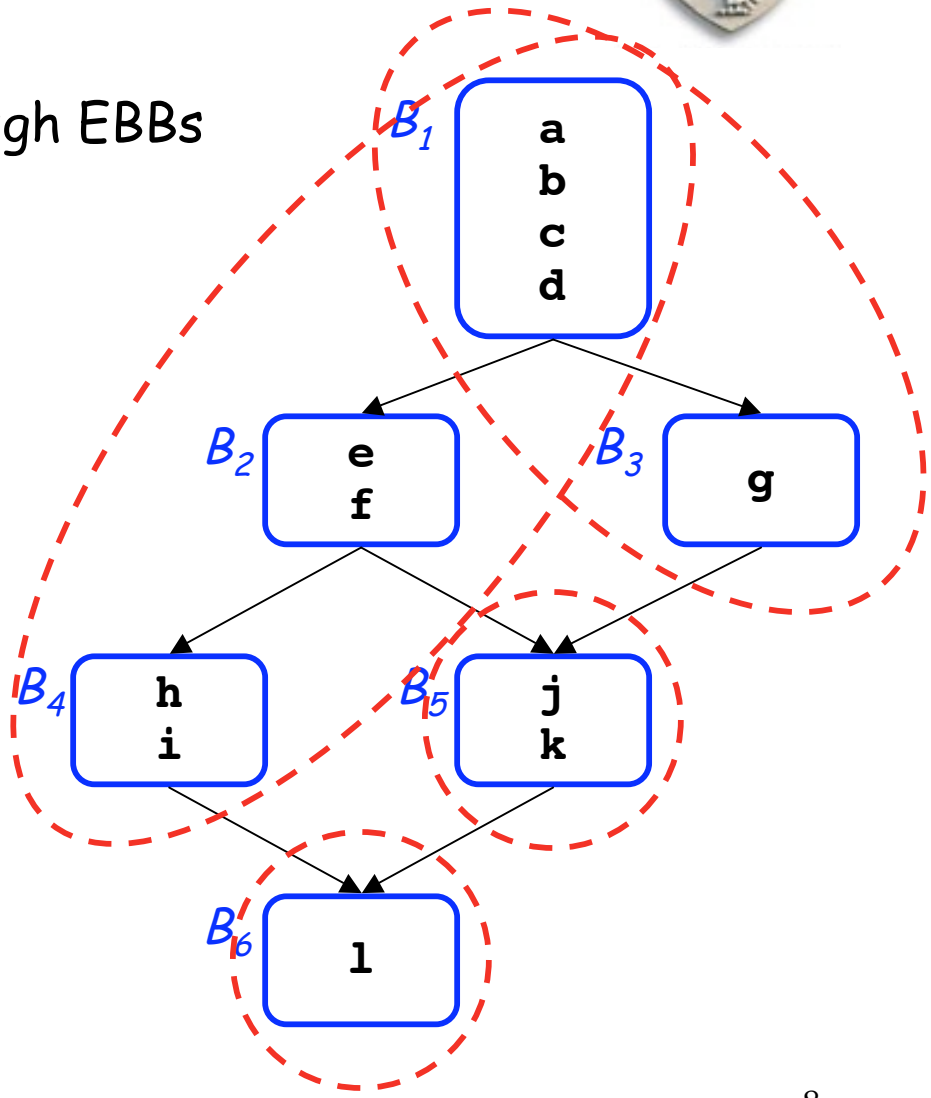
- EBB is a maximal set of blocks s.t.
  - Set has a single entry, B_i
  - Each block B_j other than B_i has exactly one predecessor
- Example has three EBBs
  - Big EBB has two paths
  - $\{B_1,B_2,B_4\}$ & $\{B_1,B_3\}$
- Many optimizations operate on EBBs (including scheduling)



Scheduling Larger Regions

Superlocal Scheduling

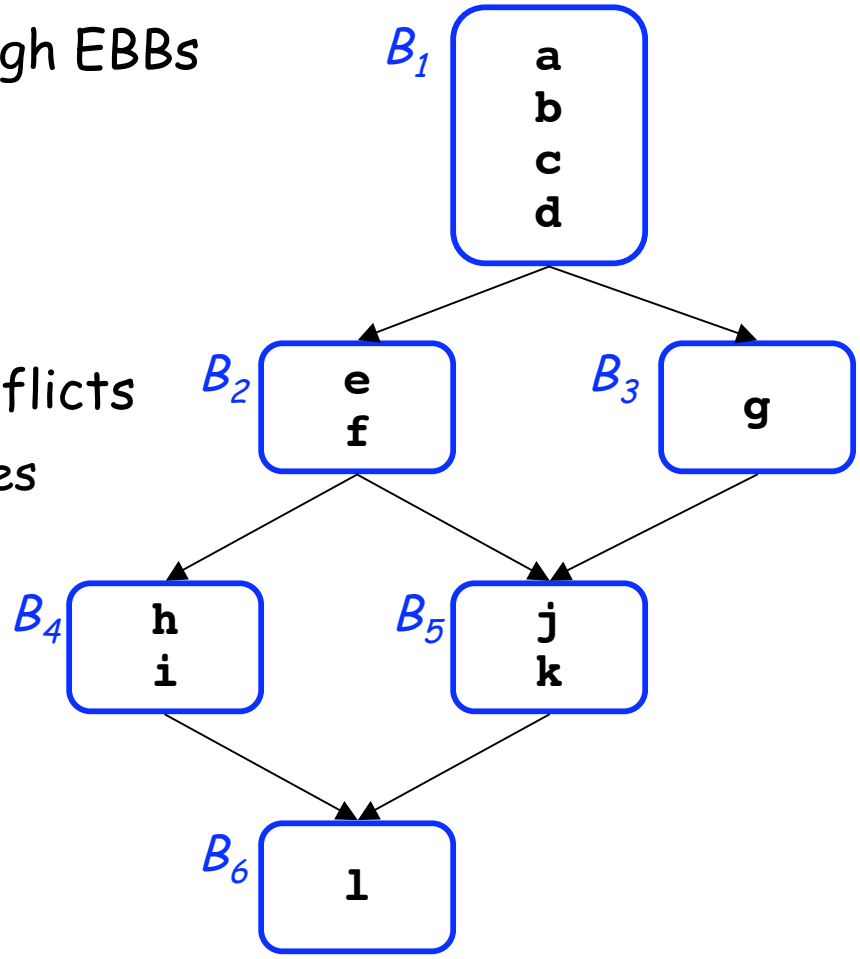
- Schedule entire paths through EBBs
- Example has four EBB paths



Scheduling Larger Regions

Superlocal Scheduling

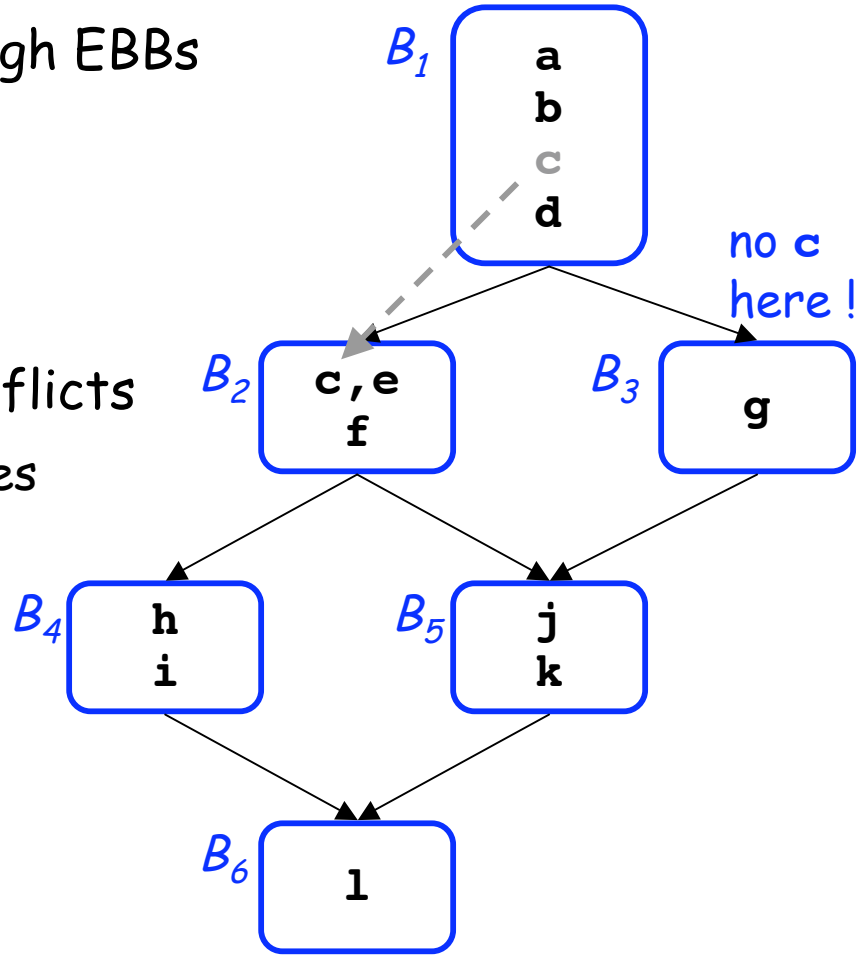
- Schedule entire paths through EBBs
- Example has four EBB paths
  - Two paths are nontrivial
  - $\{B_1,B_2,B_4\}$ & $\{B_1,B_3\}$
- Having B_1 in both causes conflicts
  - Moving an op out of B_1 causes problems



Scheduling Larger Regions

Superlocal Scheduling

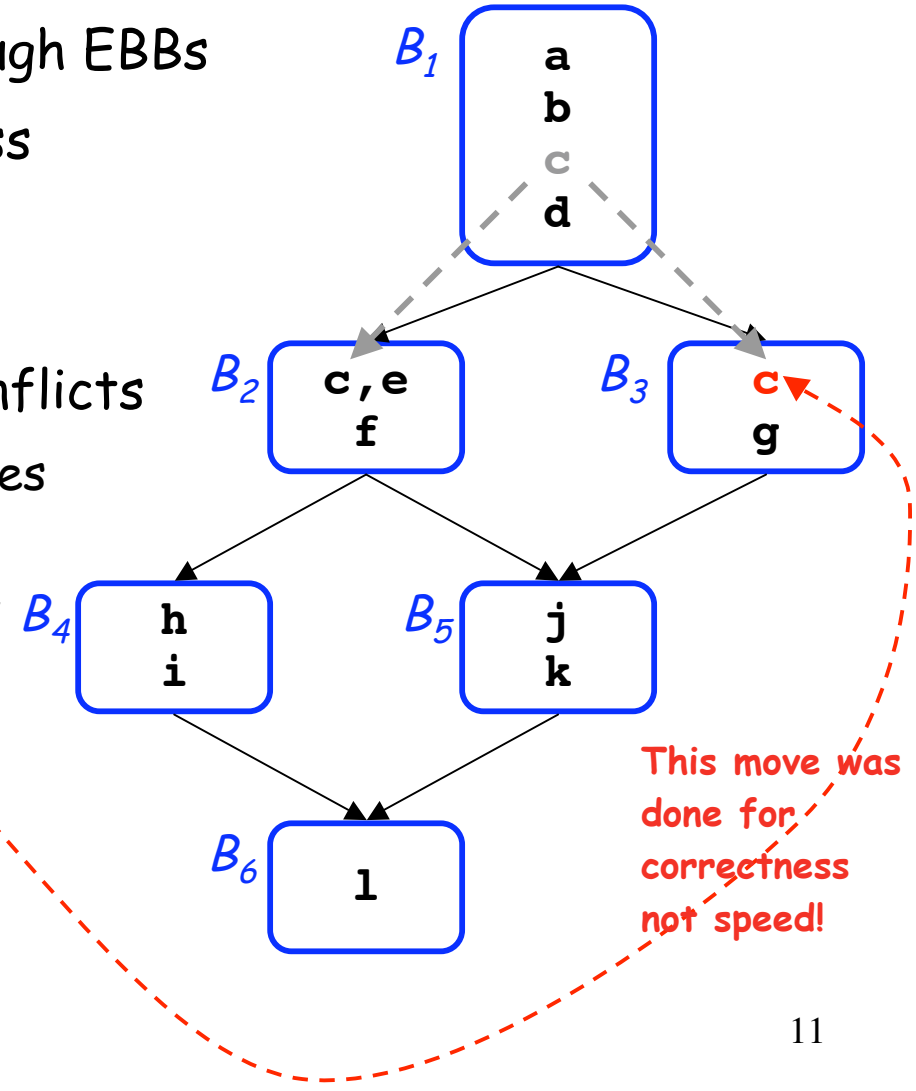
- Schedule entire paths through EBBs
- Example has four EBB paths
  - Two paths are nontrivial
  - $\{B_1,B_2,B_4\}$ & $\{B_1,B_3\}$
- Having B_1 in both causes conflicts
  - Moving an op out of B_1 causes problems



Scheduling Larger Regions

Superlocal Scheduling

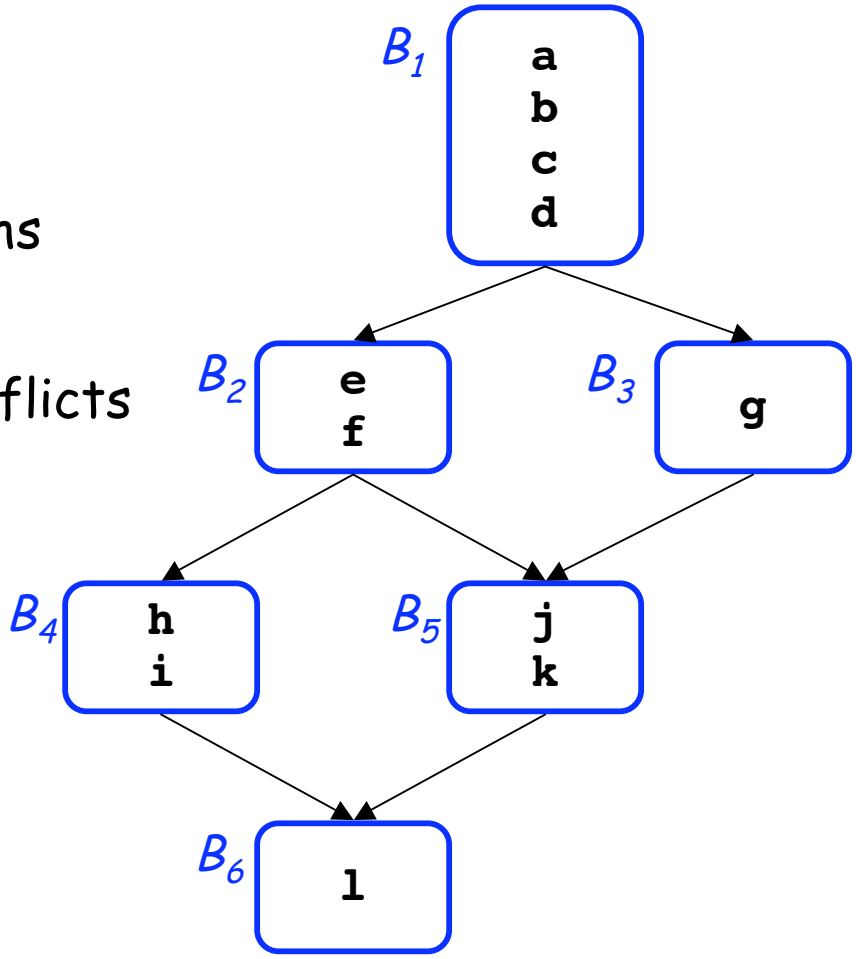
- Schedule entire paths through EBBs
- Example has four EBB pathss
  - Two paths are nontrivial
  - $\{B_1,B_2,B_4\}$ & $\{B_1,B_3\}$
- Having B_1 in both causes conflicts
  - Moving an op out of B_1 causes problems
  - Must insert "compensation" code in B_3
  - Increases code space
  - May not help on $\{B_1,B_3\}$



Scheduling Larger Regions

Superlocal Scheduling

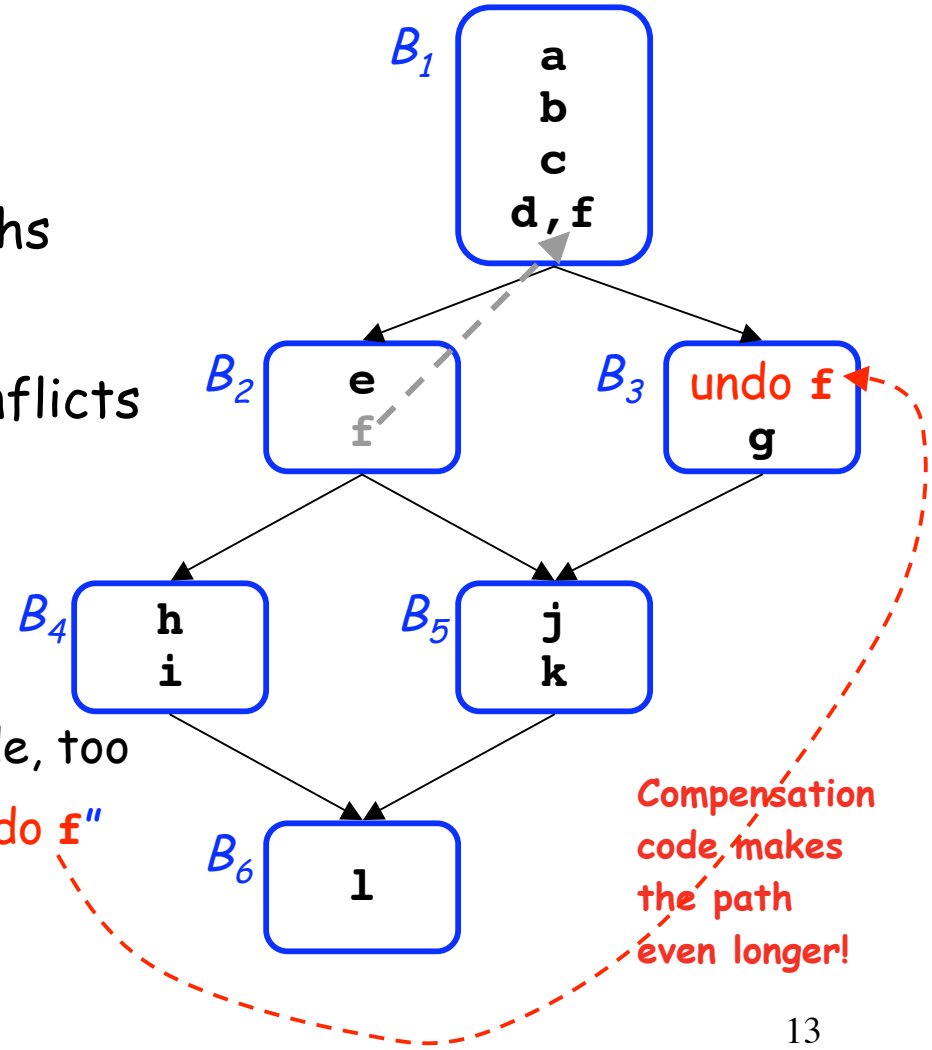
- Work EBB at a time
- Example has four EBBs
- Only two have nontrivial paths
  - $\{B_1,B_2,B_4\}$ & $\{B_1,B_3\}$
- Having B_1 in both causes conflicts
  - Moving an op into B_1 causes problems



Scheduling Larger Regions

Superlocal Scheduling

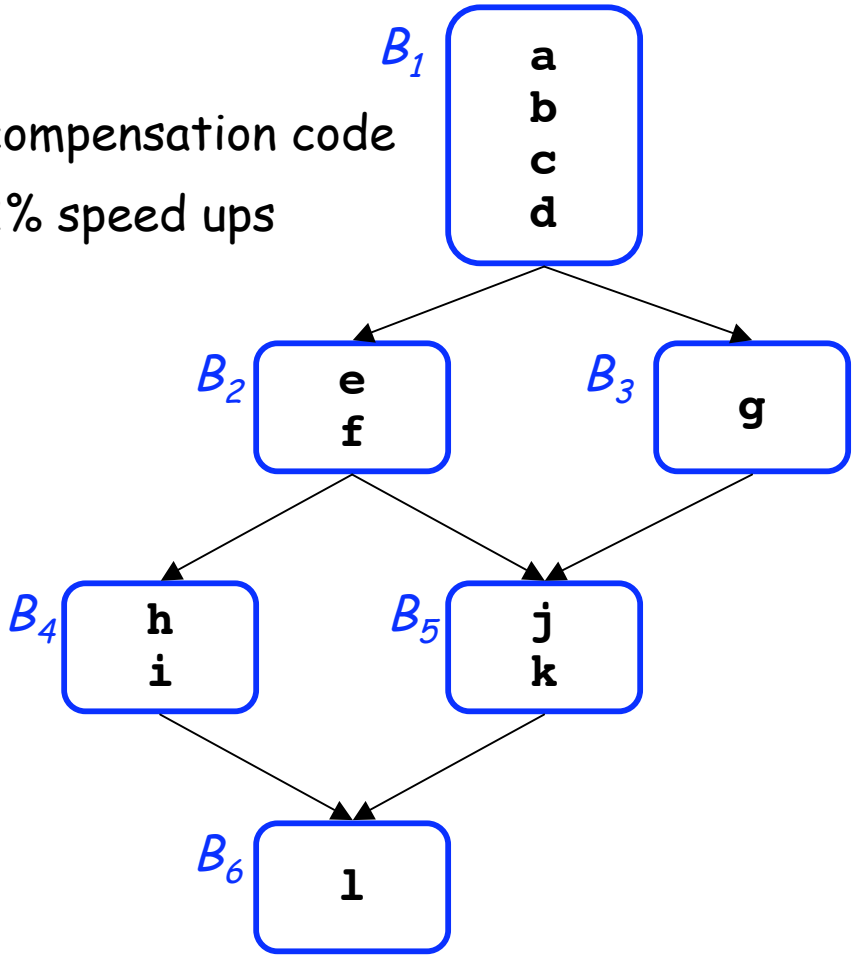
- Work EBB at a time
- Example has four EBBs
- Only two have nontrivial paths
  - $\{B_1,B_2,B_4\}$ & $\{B_1,B_3\}$
- Having B_1 in both causes conflicts
  - Moving an op into B_1 causes problems
  - Lengthens $\{B_1,B_3\}$
  - May need compensation code, too
    - Renaming may avoid "undo f"



Scheduling Larger Regions

Superlocal Scheduling

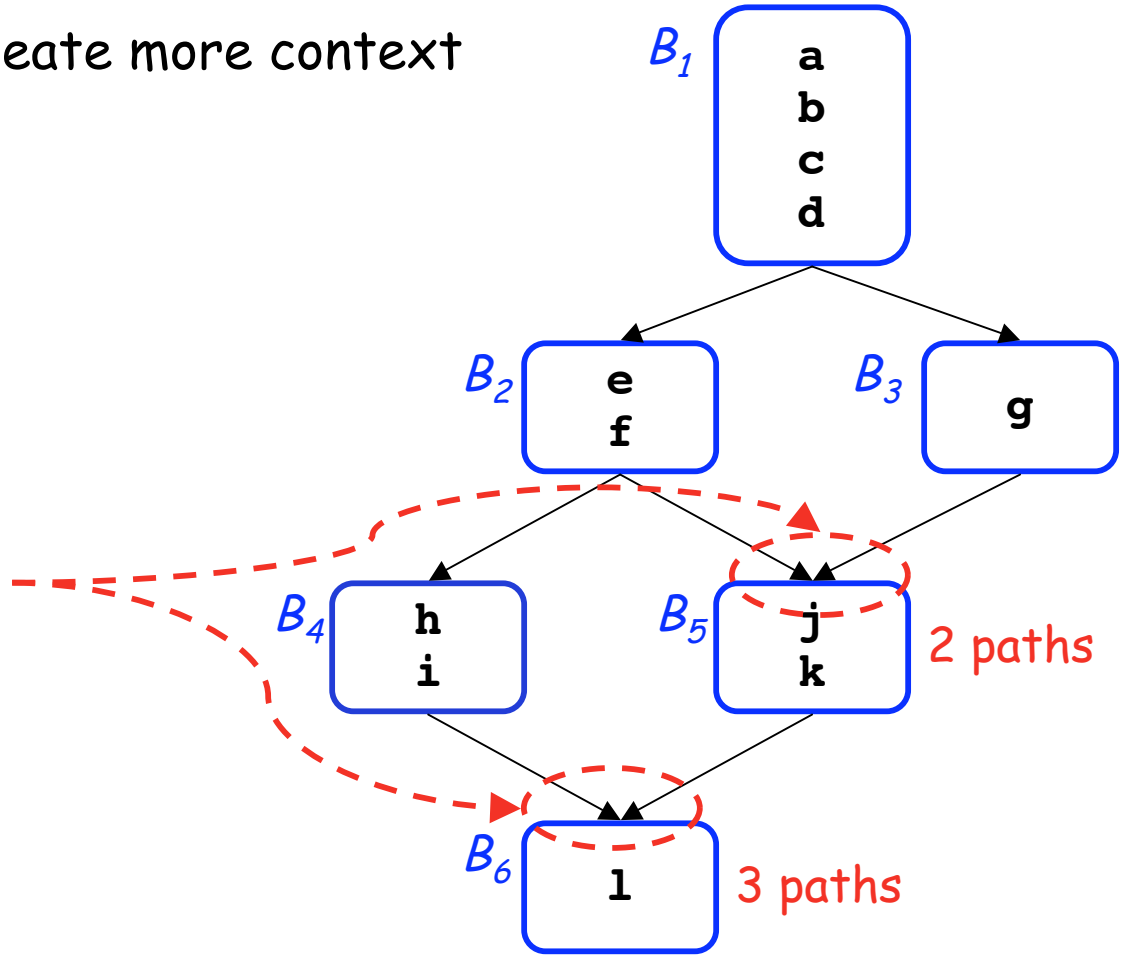
- How much can we get?
  - Schielke constrained away compensation code
  - Algorithm produced 11 to 12% speed ups
  - So, it is worth doing …



Scheduling Larger Regions

More Aggressive Superlocal Scheduling

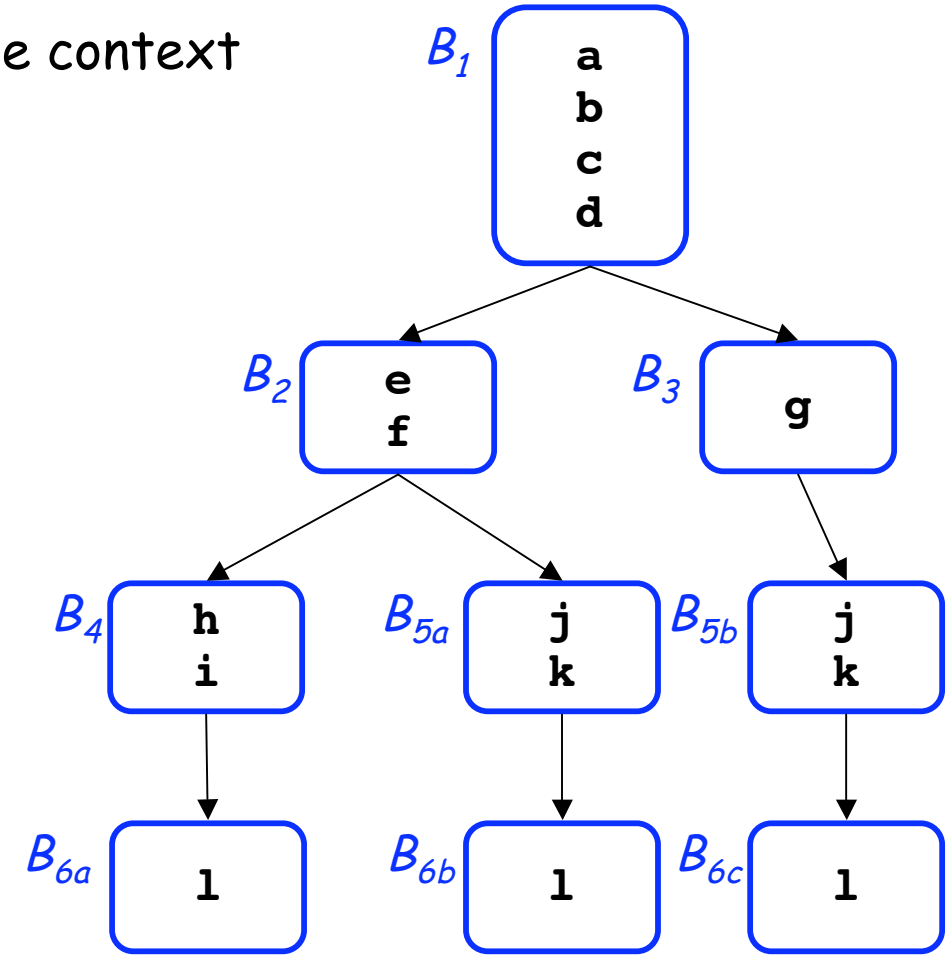
- Clone blocks to create more context



Scheduling Larger Regions

More Aggressive Superlocal Scheduling

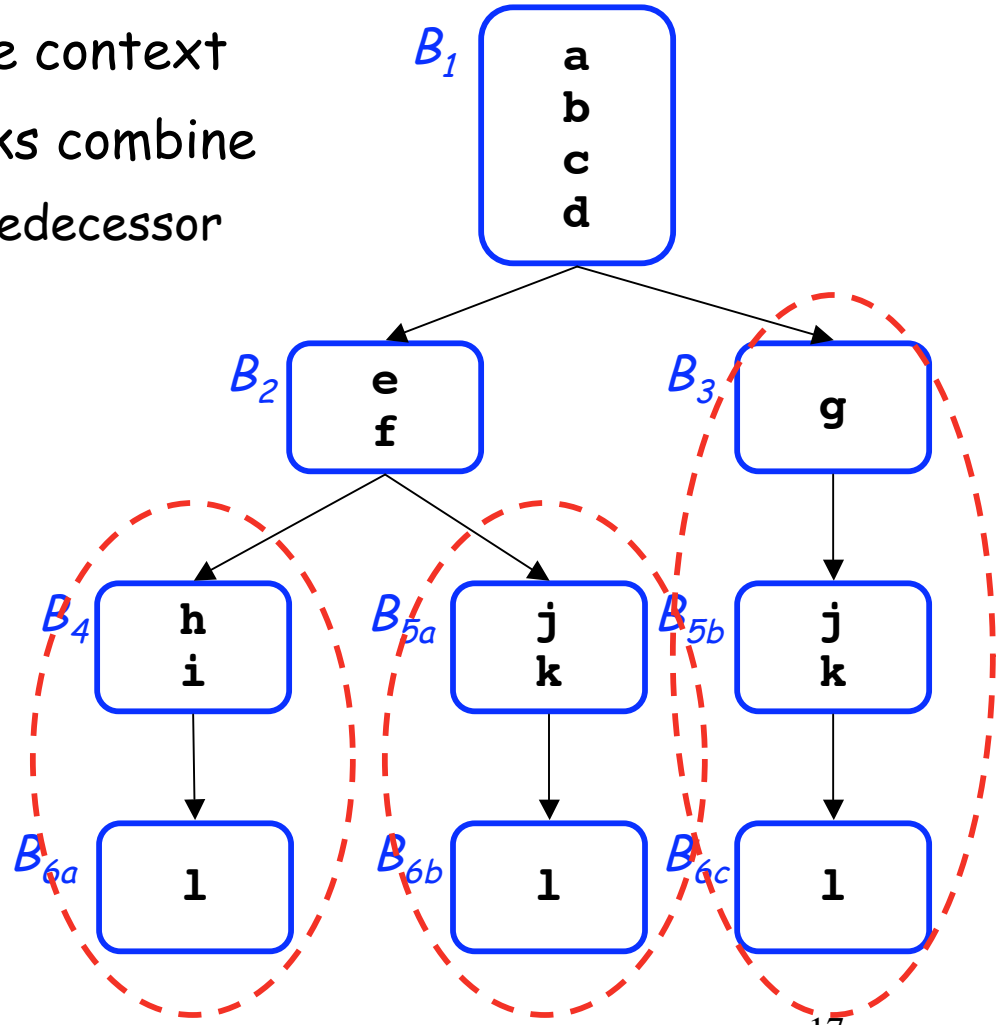
- Clone blocks to create more context



Scheduling Larger Regions

More Aggressive Superlocal Scheduling

- Clone blocks to create more context
- Some of the resulting blocks combine
  - Single successor, single predecessor

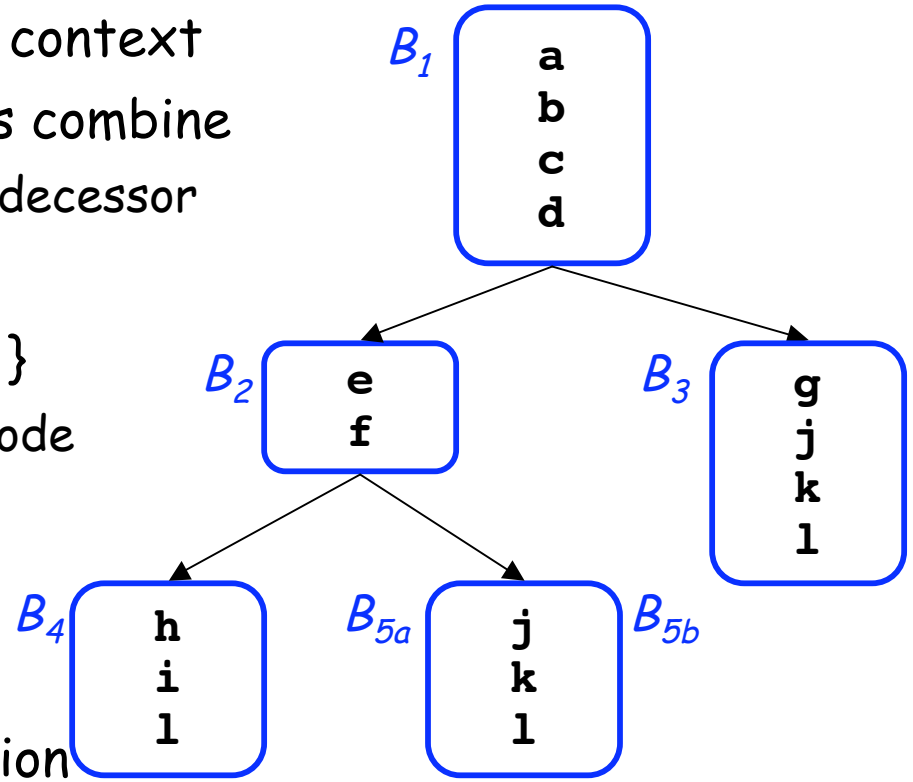


Scheduling Larger Regions

More Aggressive Superlocal Scheduling

- Clone blocks to create more context
- Some of the resulting blocks combine
  - Single successor, single predecessor
- Now schedule EBBs

  $\{B_1,B_2,B_4\}, \{B_1,B_2,B_{5q}\}, \{B_1,B_3\}$
  - Pay heed to compensation code

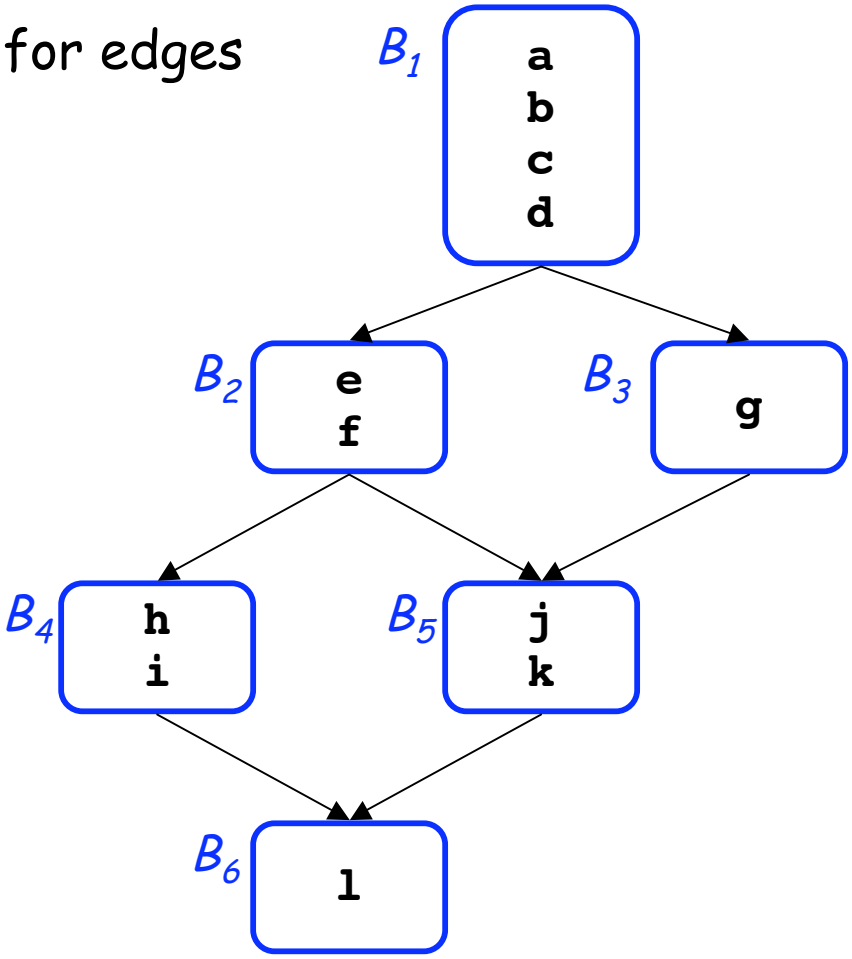


- Works well for forward motion
- Backward motion still has off-path problems
  - Speeding up one path can slow down others (*undo*)

Scheduling Larger Regions

Trace Scheduling

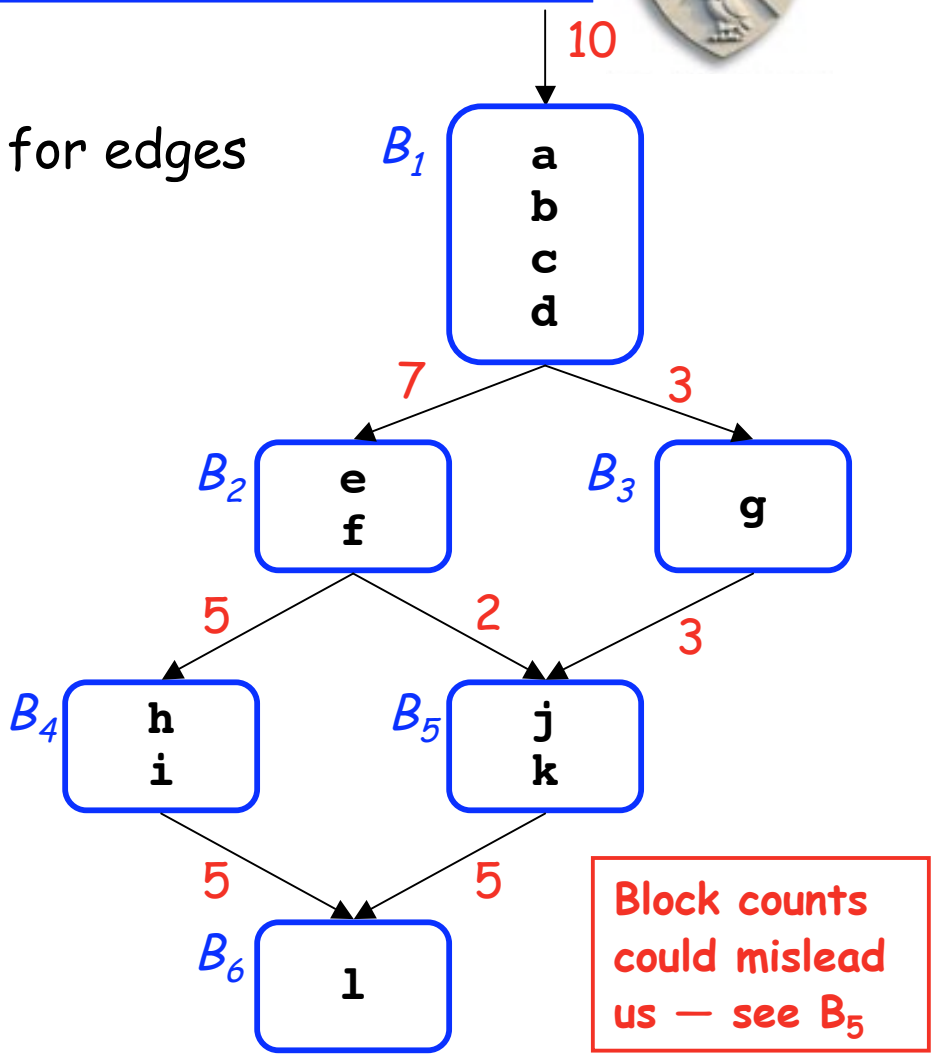
- Start with execution counts for edges
  - Obtained by profiling



Scheduling Larger Regions

Trace Scheduling

- Start with execution counts for edges
  - Obtained by profiling
- Pick the "hot" path



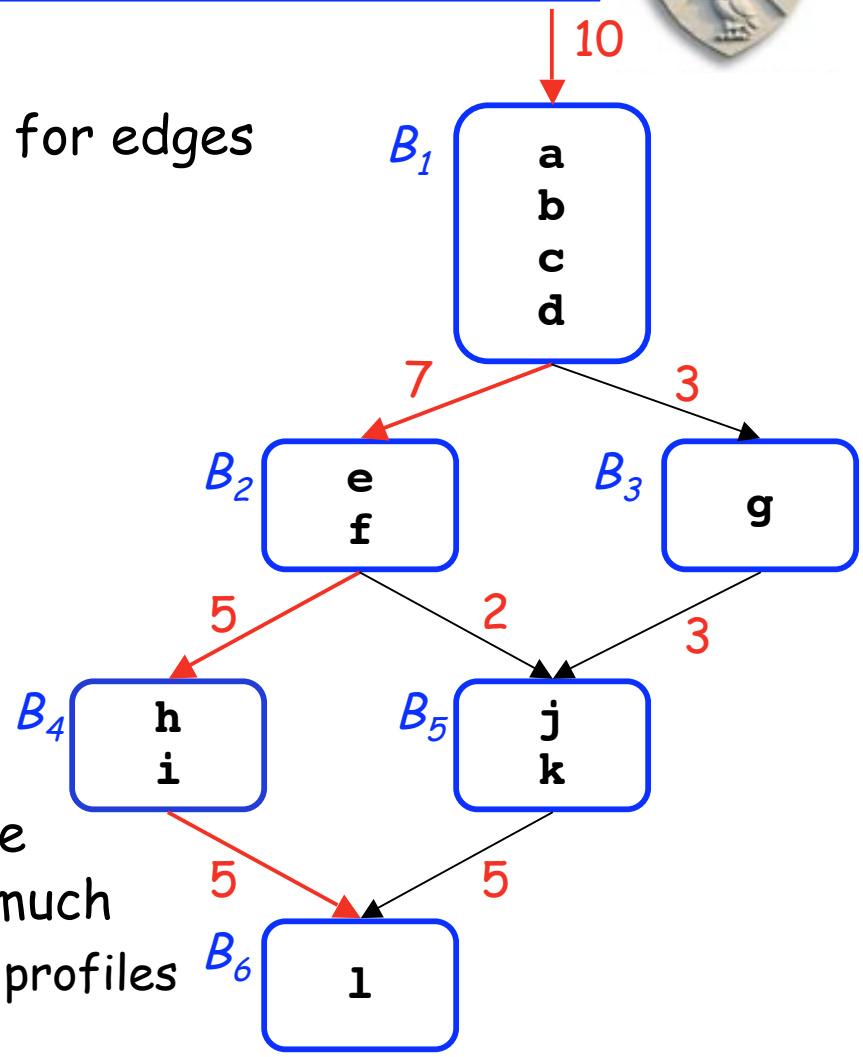
Scheduling Larger Regions

Trace Scheduling

- Start with execution counts for edges
  - Obtained by profiling
- Pick the "hot" path
  - B_1, B_2, B_4, B_6
- Schedule it
  - Compensation code in B_3, B_5 if needed
  - Get the hot path right!

If we picked the right path, the other blocks do not matter as much

- Places a premium on quality profiles



Scheduling Larger Regions

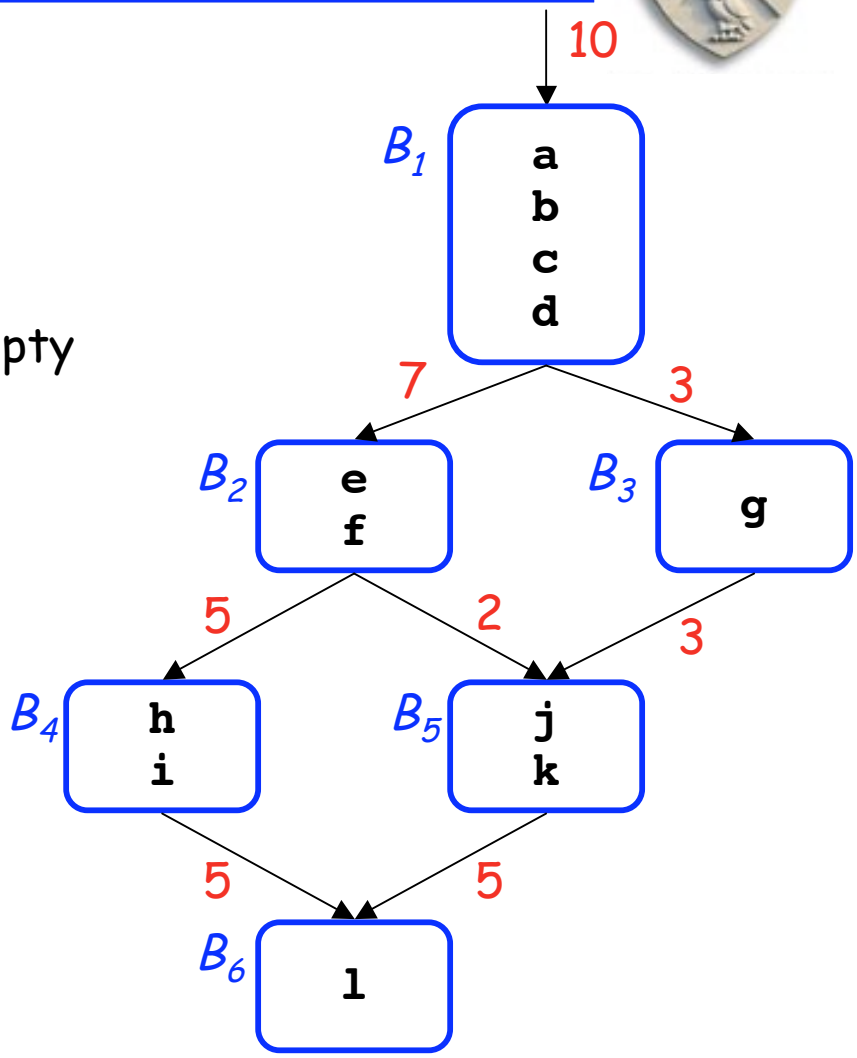
Trace Scheduling Entire CFG

- Pick & schedule hot path
- Insert compensation code
- Remove hot path from CFG

Repeat the process until CFG is empty

Idea

- Hot paths matter
- The farther we go off the hot path, the less it matters



Loop Scheduling (Software Pipelining)

Loops deserve special attention

- Their bodies execute frequently
- They do most of the work in time-critical computations
- They often contain major holes & interlocks

The ideas

- Schedule multiple iterations together
- Run several iterations concurrently
- Shorten "initiation interval" for overall speed
  - Cycles between initiation of different iterations
  - Equals the length of the computation kernel

Example

```
        loadI       0 ⇒ r1
        loadI       400 ⇒ r2
        floadAI     c ⇒ fr1
l0      floadAI     r1,a ⇒ fr2
l1      fadd        fr2,fr1 ⇒ fr2
l2      fstoreAI    fr2 ⇒ r1,b
l3      addI        r1,8 ⇒ r1
l4      cmp_LE      r1,r2 ⇒ r3
l5      cbr         r1 ⇒ l0,l6
```

2 cycle delay

3 cycle delay

How fast (in cycles per iteration) can we execute this loop?

Minimum Number of Cycles in Kernel

- Machine resource constraint:
  - N_u is the number of units of type u
  - I_u is the number of instructions requiring a unit of type u
  - The expression $\lceil I_u / N_u \rceil$ represents the minimum number of cycles required for one iteration of the loop based on unit u
  - $\max_u \lceil I_u / N_u \rceil$ is the minimum number of cycles required for all units
- Slope constraint
  - If the loop computes a recurrence with total delay of k_r cycles
  - And the number of iterations crossed by the recurrence is d_r
  - Then each iteration is going to require k_r / d_r cycles to execute
  - $\max_r \lceil k_r / d_r \rceil$ is the minimum number of cycles per iteration to compute recurrences

Example

```
        loadI       0 ⇒ r1
        loadI       400 ⇒ r2
        floadAI     c ⇒ fr1
l0      floadAI     r1,a ⇒ fr2
l1      fadd        fr2,fr1 ⇒ fr2
l2      fstoreAI    fr2 ⇒ r1,b
l3      addI        r1,8 ⇒ r1
l4      cmp_LE      r1,r2 ⇒ r3
l5      cbr         r3 ⇒ l0,l6
```

2 cycle delay

3 cycle delay

Floating Point Unit: 1 instruction
Load/Store Unit: 2 instructions
Integer Unit: 3 instructions

⇒ 3 cycles minimum

Loop Scheduling

Mechanics

- Determine lower bound on initiation interval
  - Number of issue slots
  - Longest dependence chain
- Lay out a schedule of appropriate length
- Use list scheduling with a modulo cycle count
  - Could fail — just try with one more cycle per iteration
- Add a pre-loop & a post-loop to "fill" & "drain" the pipeline

Can add a register use constraint, too

It gets pretty intricate !

- Conditional control flow complicates matters even more

Example

```
        loadI       0 ⇒ r1
        loadI       400 ⇒ r2
        floadAI     c ⇒ fr1
l0      floadAI     r1,a ⇒ fr2
l1      fadd        fr2,fr1 ⇒ fr3
l2      fstoreAI    fr3 ⇒ r1,b
l3      addI        r1,8 ⇒ r1
l4      cmp_LE      r1,r2 ⇒ r3
l5      cbr         r3 ⇒ l0,l6
```

2 cycle delay

3 cycle delay

Load/Store Unit	Integer Unit	Floating Point Unit
`floadAI r1,a ⇒ fr2`	`addI r1,8 ⇒ r1`	
	`cmp_LE r1,r2 ⇒ r3`	
`fstoreAI fr3 ⇒ r1,b-16`	`cbr r3 ⇒ l0,l6`	`fadd fr2,fr1 ⇒ fr3`

Final Code

```
   loadI     0 ⇒ r1
   loadI     400 ⇒ r2
   floadAI   c ⇒ fr1
p1 floadAI   r1,a ⇒ fr2;    addI     r1,8 ⇒ r1
p2                          cmp_LE   r1,r2 ⇒ r3
p3                          cbr      r3 ⇒ k1,e1;    fadd fr1,fr2 ⇒ fr3
k1 floadAI   r1,a ⇒ fr2;    addI     r1,r1,8
k2                          cmp_LE   r1,r2 ⇒ r3
k3 fstoreAI  fr3⇒ r1,b-16;  cbr      r3 ⇒ k1,e1;    fadd fr1,fr2 ⇒ fr3
e1 nop
e2 nop
e3 fstoreAI  fr3⇒ r1,b-8
```