CS 335: Instruction Scheduling

Swarnendu Biswas

Semester 2019-2020-II

CSE, IIT Kanpur

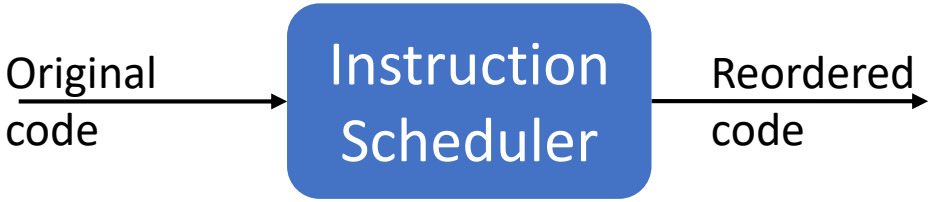
Content influenced by many excellent references, see References slide for acknowledgements.

Sensitivity to Instruction Order

- Order of instruction execution has a significant effect on program performance
  - Different operations have different latencies
  - Same operation may have different latencies

- Instruction scheduling is the task of ordering the operations to make effective use of processor resources
  - Input to the instruction scheduler is an unordered or partially ordered list of operations in say the target machine's assembly language
  - Output is an order on the list of operations

Instruction Scheduling



- Compiler reorders operations in the compiled code in an attempt to decrease its running time
- Scheduler assumes a fixed set of operations and does not rewrite code
  - May add nops to maintain dependence
- Scheduler assumes a fixed allocation of values to registers
  - May rename registers but does not change allocation decisions
  - Should avoid increasing the lifetime of values since it may lead to register spills

Overlapping Instructions

- Processors overlap instruction execution to make use of a finite set of functional units
- Processor stalls an instruction until its operands are available
  - Scheduler can reorder instructions to minimize the number of stalls
- Processor can also continue executing the instruction with wrong operands
  - Will need support for re-execution when correct operands are available
  - Need to maintain sufficient distance between the definition and the uses of the operand

Issuing Instructions

- Many processors can issue multiple operations per cycle
  - Superscalar processor can issue distinct operations to multiple distinct functional units in a single cycle
  - VLIW processor issue an operation for each functional unit in each cycle
- Superscalar processors
  - Monitor a small window in the instruction stream
  - Choose operations that can execute on available units
  - Assign ready operations to functional units.
- Window size is relatively larger for out-of-order superscalar processors

Instruction Scheduling

- A processor that relies on the compiler to insert NOPs for correctness is called a statically scheduled processor
  - Scheduler checks the availability of functional units
- A processor that uses interlocks to ensure correctness is a dynamically scheduled processor
  - An interlock is a hardware mechanism to detect premature issue and introduces a delay
  - Scheduler checks availability of operands

Instruction Scheduling Example

Start	Operations
1	LOAD R_{ARP}, @a $\Rightarrow R_1$
4	ADD $R_1, R_1 \Rightarrow R_1$
5	LOAD R_{ARP}, @b $\Rightarrow R_2$
8	MUL $R_1, R_2 \Rightarrow R_1$
10	LOAD R_{ARP}, @c $\Rightarrow R_2$
13	MUL $R_1, R_2 \Rightarrow R_1$
15	LOAD R_{ARP}, @d $\Rightarrow R_2$
18	MUL $R_1, R_2 \Rightarrow R_1$
20	STORE $R_1 \Rightarrow R_{ARP}$, @a

Start	Operations
1	LOAD R_{ARP}, @a $\Rightarrow R_1$
2	LOAD R_{ARP}, @b $\Rightarrow R_2$
3	LOAD R_{ARP}, @c $\Rightarrow R_3$
4	ADD $R_1, R_1 \Rightarrow R_1$
5	MUL $R_1, R_2 \Rightarrow R_1$
6	LOAD R_{ARP}, @d $\Rightarrow R_2$
7	MUL $R_1, R_3 \Rightarrow R_1$
9	MUL $R_1, R_2 \Rightarrow R_1$
11	STORE $R_1 \Rightarrow R_{ARP}$, @a

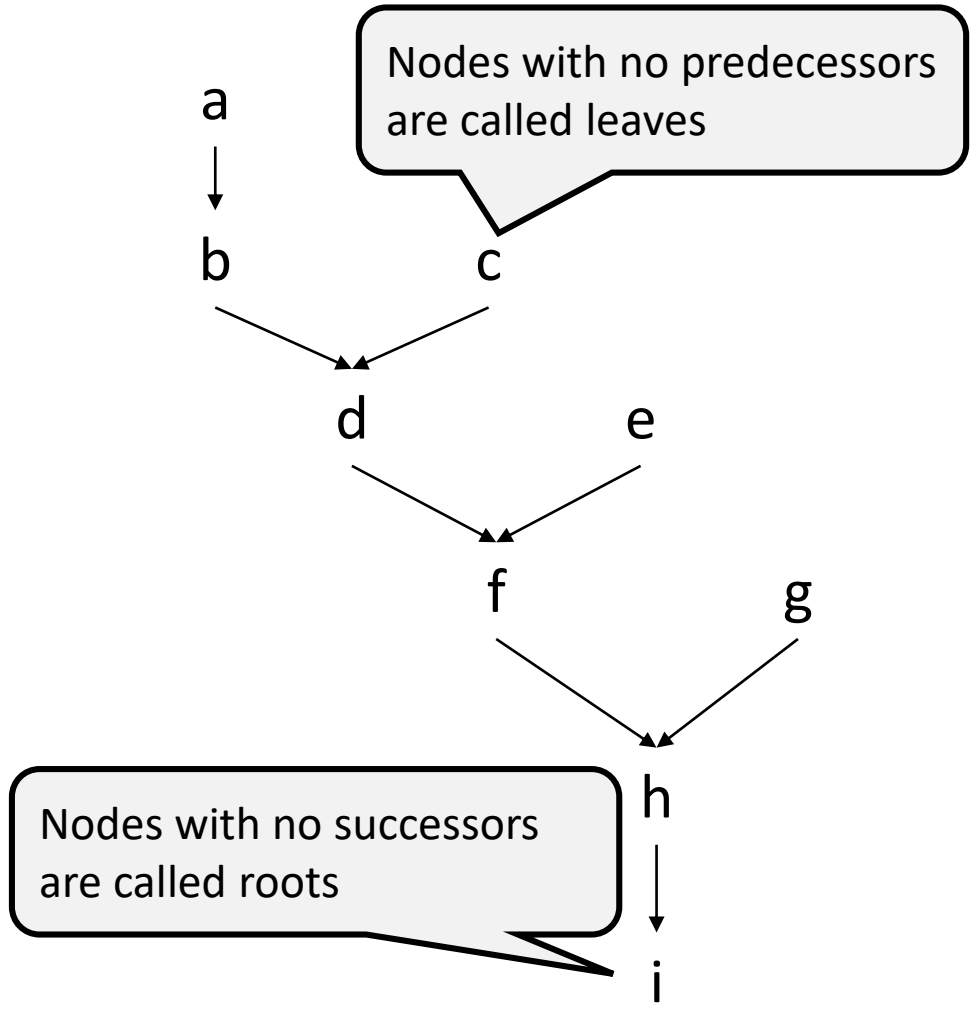
Dependence Graph

- Given a basic block B, its dependence graph is $D = (N, E)$
  - D has a node for each operation in B
  - An edge in D connects two nodes n_1 and n_2 if n_2 uses the result of n_1
  - Edges represent flow of values
  - D is also called a precedence graph

- Each node n has two attributes
  - Operation type – functional unit on which the operation must execute
  - Delay – number of cycles to complete

Example of a Dependence Graph

Start	Operations	Symbol
1	LOAD R_{ARP}, @a $\Rightarrow R_1$	a
4	ADD $R_1, R_1 \Rightarrow R_1$	b
5	LOAD R_{ARP}, @b $\Rightarrow R_2$	c
8	MUL $R_1, R_2 \Rightarrow R_1$	d
10	LOAD R_{ARP}, @c $\Rightarrow R_2$	e
13	MUL $R_1, R_2 \Rightarrow R_1$	f
15	LOAD R_{ARP}, @d $\Rightarrow R_2$	g
18	MUL $R_1, R_2 \Rightarrow R_1$	h
20	STORE $R_1 \Rightarrow R_{ARP}$, @a	i

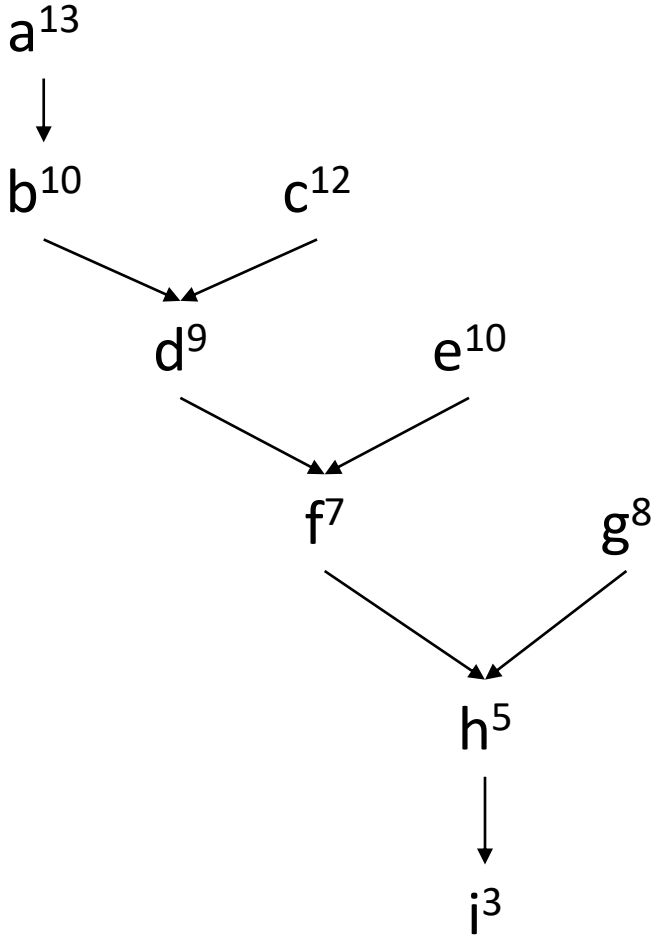


Instruction Scheduling

- A schedule S maps each node $n \in N$ to a nonnegative integer that denotes the cycle in which it should be issued
- An instruction i can have multiple operations
  - Operations are denoted by $\{ n \mid S(n) == i \}$
- Constraints
  i. $S(n) \geq 1$, with at least one operation n' such that $S(n') = 1$
  ii. If $(n_1, n_2) \in E$, then $S(n_1) + delay(n_1) \leq S(n_2)$
  iii. Each instruction contains no more operations of each type than the target machine can issue in a cycle

Instruction Scheduling

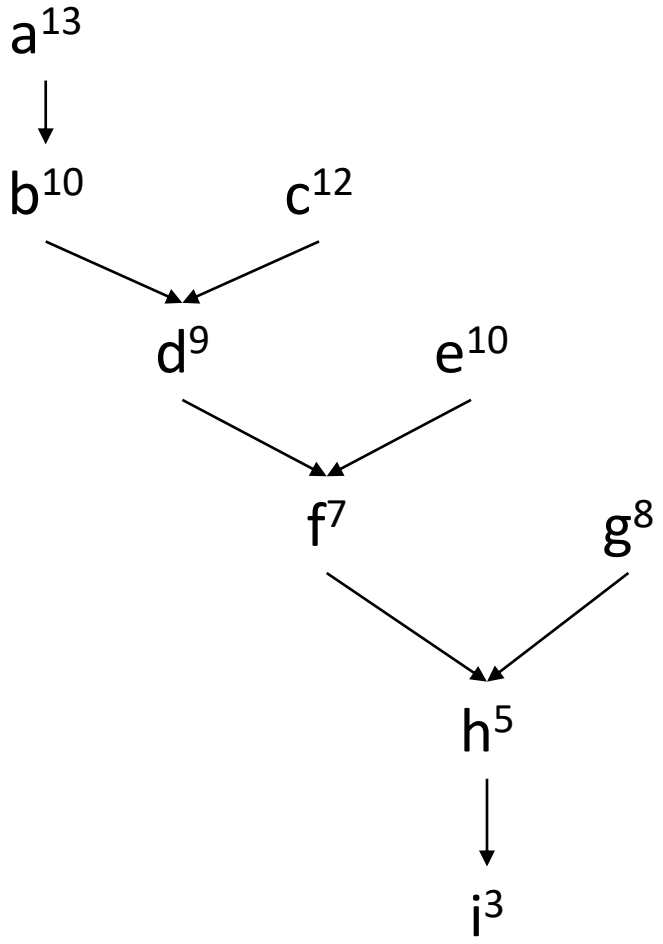
- Given a well-formed schedule, the length of the schedule is the cycle number in which the last operation completes
- Assuming first instruction is issued at cycle 1, schedule length is $L(S) = \max_{n \in N}(S(n) + delay(n))$
- A schedule S_i is time optimal if $L(S_i) \leq L(S_j) \ \forall \ S_j \neq S_i$
- Critical path is the longest latency path through D



Instruction Scheduling

- a is on the critical path, so we should schedule a first
- c can be the next, since it now lies on the longest path
- Better to schedule e before b

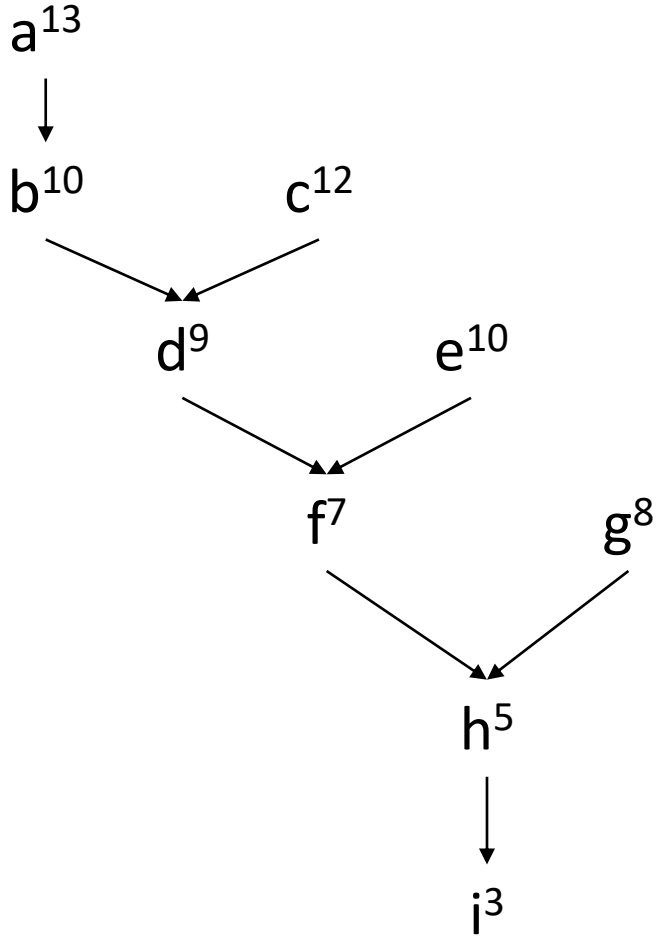
Possible schedule = ???



Instruction Scheduling

- a is on the critical path, so we should schedule a first
- c can be the next, since it now lies on the longest path
- Better to schedule e before b

Possible schedule = `acebdgfhi`



Instruction Scheduling

Start	Operations	Symbol
1	LOAD R_{ARP}, @a $\Rightarrow R_1$	a
2	LOAD R_{ARP}, @b $\Rightarrow R_2$	c
3	LOAD R_{ARP}, @c $\Rightarrow R_2$	e
4	ADD $R_1, R_1 \Rightarrow R_1$	b
5	MUL $R_1, R_2 \Rightarrow R_1$	d
6	LOAD R_{ARP}, @d $\Rightarrow R_2$	g
7	MUL $R_1, R_3 \Rightarrow R_1$	f
9	MUL $R_1, R_2 \Rightarrow R_1$	h
11	STORE $R_1 \Rightarrow R_{ARP}$, @a	i

Possible schedule = `acebdgfhi?`

Instruction Scheduling

Start	Operations	Symbol
1	LOAD R_{ARP}, @a $\Rightarrow R_1$	a
2	LOAD R_{ARP}, @b $\Rightarrow R_2$	c
3	LOAD R_{ARP}, @c $\Rightarrow R_2$	e
4	ADD $R_1, R_1 \Rightarrow R_1$	b
5	MUL $R_1, R_2 \Rightarrow R_1$	d
6	LOAD R_{ARP}, @d $\Rightarrow R_2$	g
7	MUL $R_1, R_3 \Rightarrow R_1$	f
9	MUL $R_1, R_2 \Rightarrow R_1$	h
11	STORE $R_1 \Rightarrow R_{ARP}$, @a	i

Possible schedule = `acebdgfhi?`

- Both c and e define R_2, and d uses R_2
- Compiler cannot move e before d without renaming

Dealing with Antidependence

- Operation x is antidependent on operation y if x precedes y and y defines a value used in x
  - Reversing their order of execution could cause x to compute a different value
- How can a scheduler can deal with antidependences?
  - Identify antidependences and respect the constraints in the generated schedule
    - Restricts the number of possible schedules a compiler can generate
  - Rename values to avoid antidependences
    - Increasing variable lifetime can lead to register spills

Limitations in Scheduling

- Adjacent code has dependences that cannot be avoid during scheduling

- Earlier passes may refactor the code to expose parallelism
  - $(a^2)^2 \times (a^2)^2$
  - Can proceed in parallel if there are more than one multiplication unit

Start	Operations
1	LOAD R_{ARP}, @a $\Rightarrow R_1$
4	MUL $R_1, R_1 \Rightarrow R_1$
6	MUL $R_1, R_1 \Rightarrow R_1$
8	MUL $R_1, R_1 \Rightarrow R_1$
10	MUL $R_1, R_1 \Rightarrow R_1$
12	STORE $R_1 \Rightarrow R_{ARP}$, @x

Challenges in Scheduling

- Scheduler needs to find a mapping between one or more operations and the clock cycle when they can start
  - A choice influences all reachable nodes

- Instruction scheduling is NP-complete

List Scheduling

List Scheduling

- Greedy, heuristic-based approach to schedule operations in a basic block
- Steps in applying list scheduling
  i. Rename values to avoid antidependences
  ii. Build a dependence graph D
  iii. Assign priorities to each operation
  iv. Iteratively select an operation for scheduling

Steps in List Scheduling

i. Rename values to avoid antidependences (optional step)
   - Each definition receives a unique name
   - Allows the scheduler more flexibility in identifying schedules

ii. Build a dependence graph D
   - Scheduler traverses the block from bottom to top
   - Each node represents a new value
   - Each edge is annotated with the latency of the current operation

Steps in List Scheduling

iii. Assign priorities to each operation
- Scheduler computes several different scores for each node
  - Length of the longest latency-weighted path from the node to a root in D
- Priorities are used for ordering and breaking ties

iv. Iteratively select an operation for scheduling
- Start in the first cycle for the basic block
- At each cycle, choose as many operations as possible to issue

List Scheduling Algorithm

$cycle = 1$

$Ready =$ { leaves of D }

$Active = \{\phi\}$

while $Ready \cup Active \neq \phi$

  for each $op \in Active$

    if $S(op) + delay(op) < cycle$

      $Active = Active - op$

        for each successor s of op

          if s is $Ready$

            $Ready = Ready \cup s$

if $Ready \neq \phi$

  $Ready = Ready - op$

  $S(op) = cycle$

  add op to $Active$

$cycle = cycle + 1$

List Scheduling Algorithm

- At each time step
  - the algorithm accounts for operations completed in the previous cycle
  - schedules an operation for the current cycle
  - increments $cycle$
- Block-ending jump must be scheduled such that it does not modify the program counter
  - Two-cycle branch must not be scheduled earlier than the penultimate cycle
  - If i is the block-ending branch, it cannot be scheduled earlier than cycle $L(S) + 1 - delay(i)$

Thoughts on the List Scheduling Algorithm

- If $|Ready| = 1$, then the generated schedule must be optimal
- If $|Ready| > 1$, then operation with highest priority should be chosen

Scheduling Operations with Variable Delays

- Memory operations often have variable delays
  - Assuming worst-case delay can keep the processor idle
  - Assuming best-case delay will require stalls on a cache miss
- Compilers follow balanced scheduling
  - Calculate individual latency for each load based on the amount of instruction-level parallelism available to cover the load's latency
  - Schedule the load considering the surrounding code
  - Distribute the available parallelism across the loads in the block

Computing Delays for Load Operations

```
for each load operation l in the block
  delay(l) = 1
for each operation i in D
  let D_i be the nodes and edges in D independent of i

  for each connected component C of D_i do
    find the maximal number of loads N on any path through C

    for each load operation l in C
      delay(l) = delay(l) + delay(i)/N
```

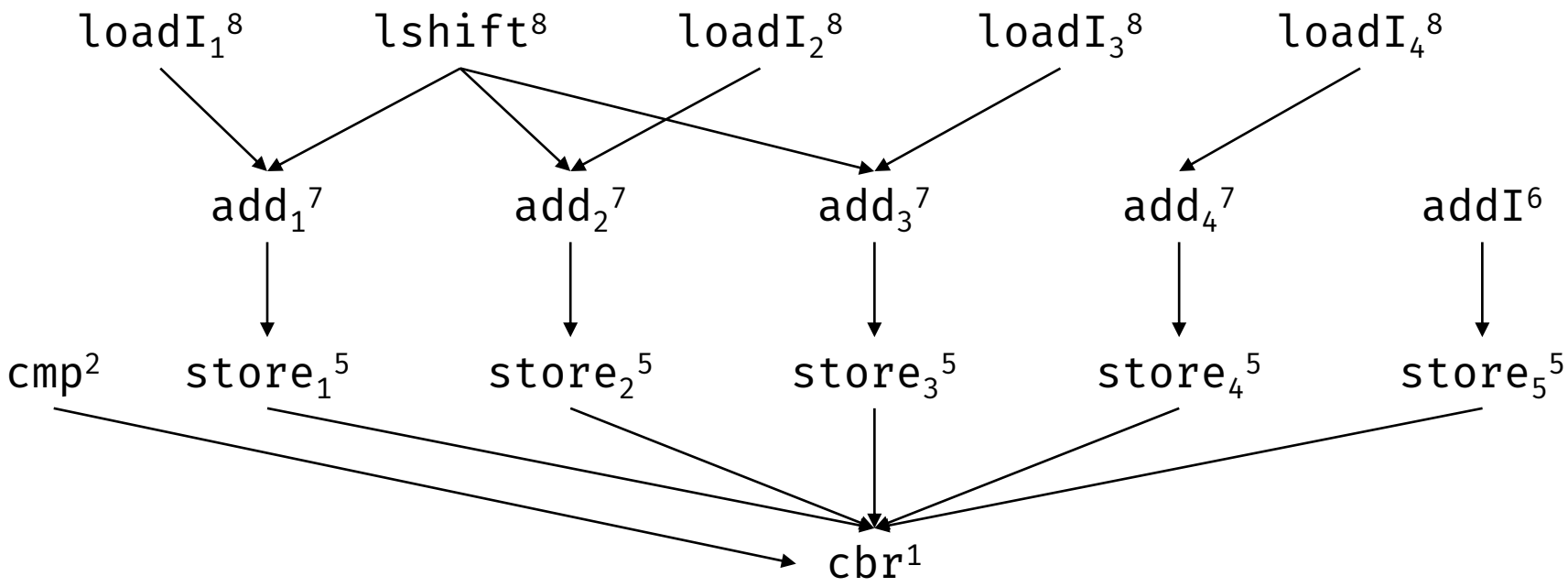
Other Considerations

- Algorithm assumes only one operation is issued per cycle
  - The algorithm should consider one operation per functional unit per cycle
- Some operations can execute on multiple functional units while others cannot
  - Schedule the more-constrained units before the less-constrained units
- Operands computed in predecessor blocks may not be available during the first cycle at block boundaries

Other Priority Measures for Tie Breaking

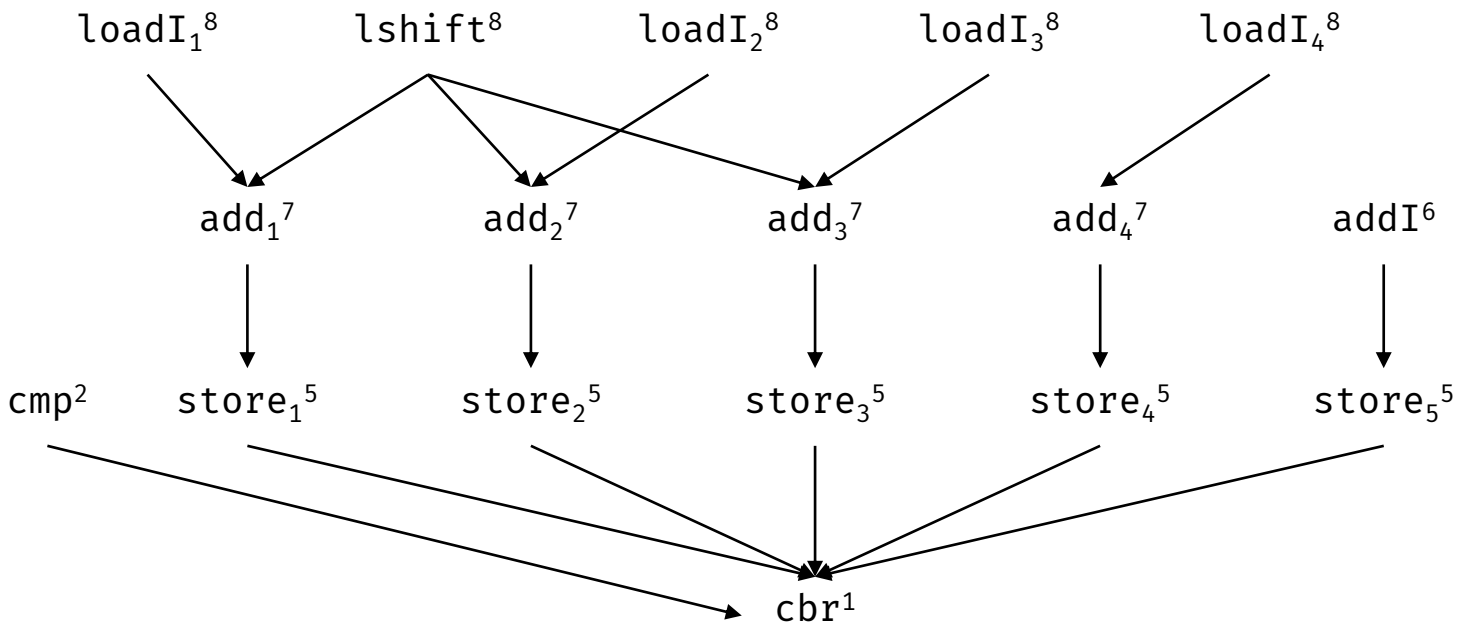
- A node's rank is the number of immediate successors it has in D
  - Encourages the scheduler to pursue many distinct paths through D, similar to a breadth-first approach
- A node's rank is the total number of descendants it has in D
- A node's rank is equal to its $delay$
  - Schedules long-latency operations as soon as possible
- A node's rank is equal to the number of operands for which this operation is the last use
  - Moves last uses closer to their definitions to decrease demand for registers

Example of Forward List Scheduling



Opcode	loadI	lshift	add	addI	cmp	store
Latency	1	1	2	1	1	4

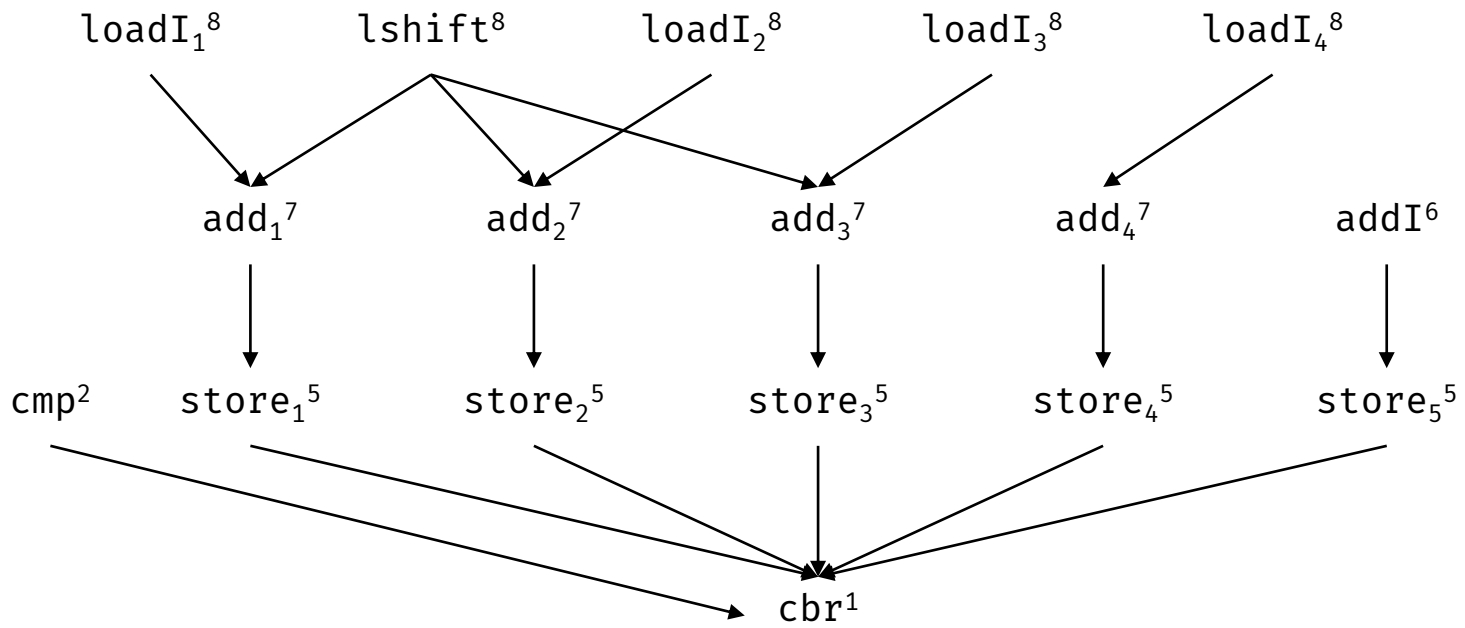
Example of Forward List Scheduling



	Integer	Integer	Memory
1			
2			
3			
4			
5			
6			
7			

Opcode	loadI	lshift	add	addI	cmp	store
Latency	1	1	2	1	1	4

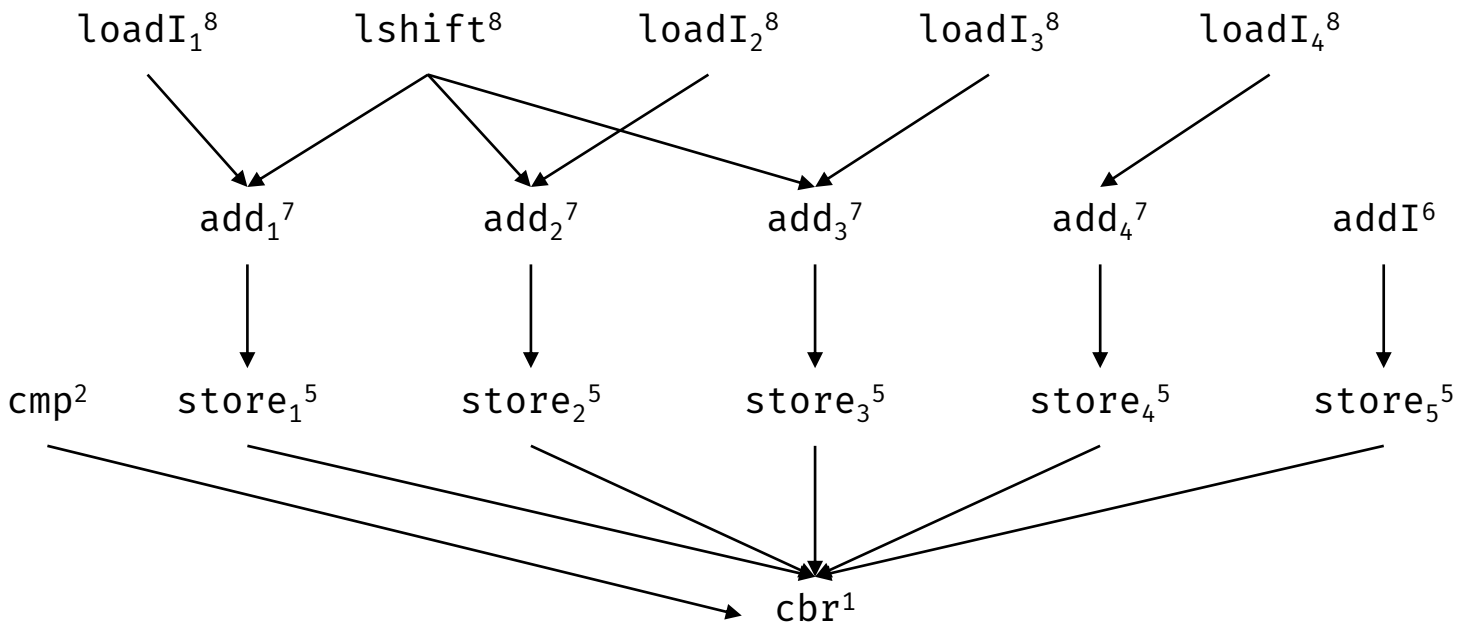
Example of Forward List Scheduling



Opcode	loadI	lshift	add	addI	cmp	store
Latency	1	1	2	1	1	4

	Integer	Integer	Memory
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		

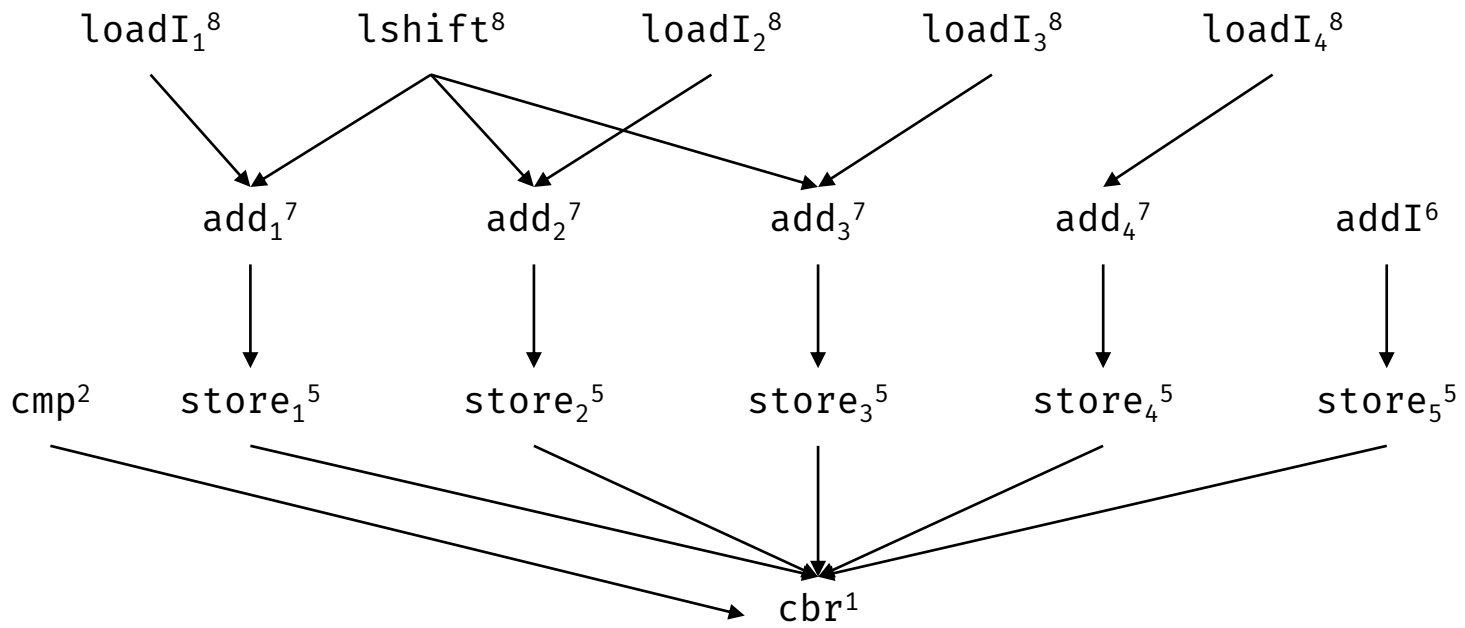
Example of Backward List Scheduling



Opcode	loadI	lshift	add	addI	cmp	store
Latency	1	1	2	1	1	4

	Integer	Integer	Memory
	cmp		
	cbr		

Example of Backward List Scheduling



Opcode	loadI	lshift	add	addI	cmp	store
Latency	1	1	2	1	1	4

	Integer	Integer	Memory
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		

Does OOO Eliminate the Need for Instruction Scheduling?

- Many modern processors support out-of-order (OOO) execution
  - The dynamically-scheduled processor maintains a portion of the dependence graph at run time to identify when each instruction can execute
- When can OOO processor improve on a static schedule?
  - Run-time information is more precise than the assumptions made by the scheduler
    - An operand at a block boundary is available before its worst-case time
    - More precise estimates for variable-latency operations
    - Can precisely identify load-store dependences because the hardware knows actual runtime addresses while a static scheduler does not
  - The OOO processor might issue an operation earlier than its position in the static schedule

Does OOO Eliminate the Need for Instruction Scheduling?

- OOO execution does not eliminate the need for instruction scheduling because the lookahead window is finite
  - Consider a string of 100 integer instructions followed by 100 floating-point instructions and a lookahead window of 50 instructions It may, however,
- OOO execution helps the compiler by improving good, but nonoptimal, schedules

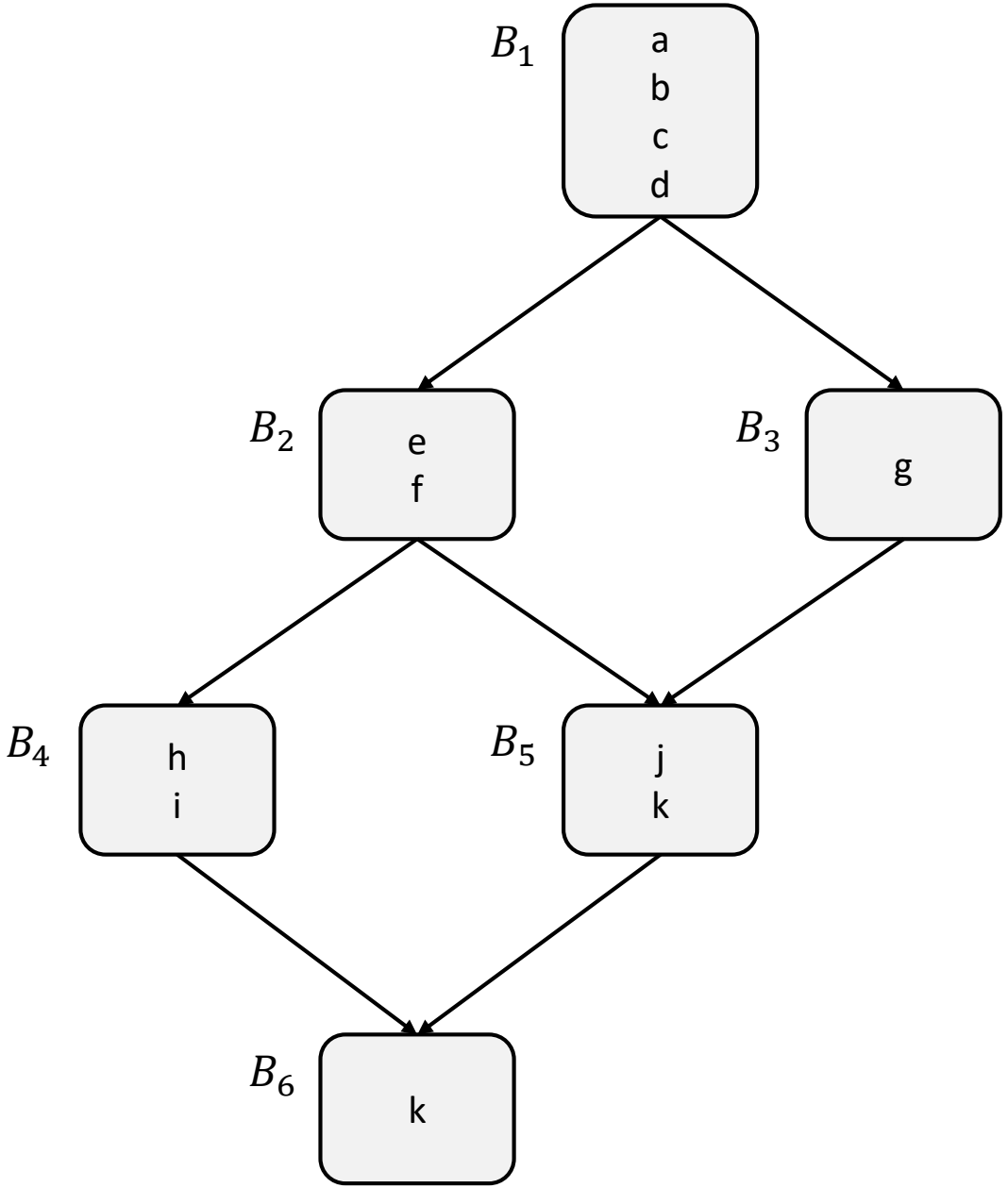
Regional Scheduling

Extending Beyond BBs

- Limiting analysis to BBs is simple and convenient
- However, extending the window of scheduling beyond BBs can improve the code quality
  - Span can be multiple BBs in a procedure
  - Goal is to increase code that can be scheduled together
- Almost all proposed ideas use the list scheduling algorithm at its core

Extended BB (EBB)

- An extended BB is a set of BBs $\{B_1, B_2, \ldots, B_n\}$ such that
  - B_1 has multiple predecessors
  - Any other block B_i has exactly one predecessor B_j in the EBB

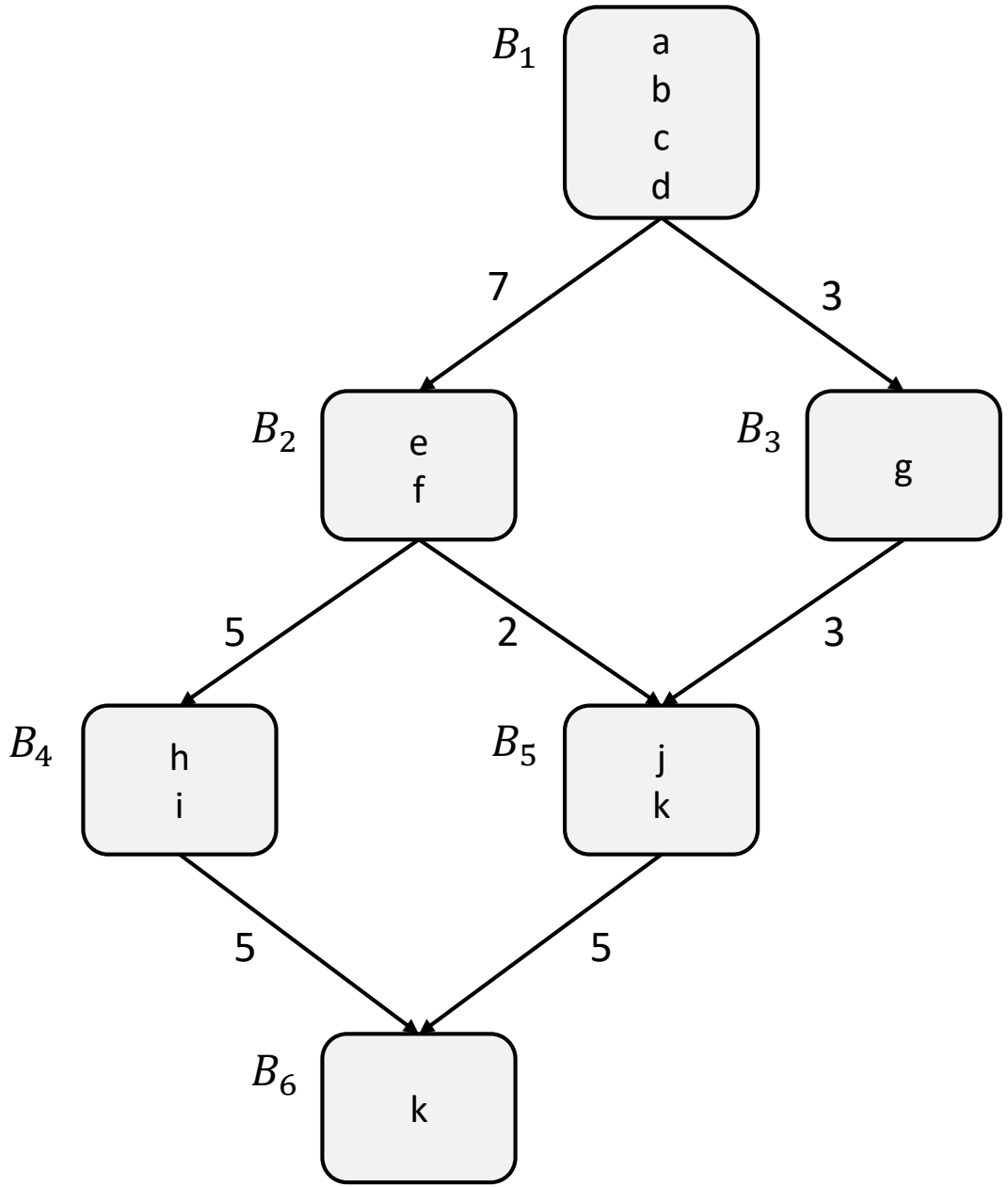


Scheduling EBBs

- Compilers process paths in an EBB for scheduling
  - For example, $\{B_1, B_2, B_4\}$ and $\{B_1, B_3\}$
- Challenges
  - Compiler must reason about any code motion performed on one path on other paths
    - Compiler can move c from B_1 to B_2 to improve the performance of the first path
    - Compiler must compensate, insert c into B_3
  - Similarly, a compiler might move f from B_2 to B_1
    - This can lead to erroneous output in the path $\{B_1, B_3\}$
    - Either rename the output of f or insert an undo operation
- Scheduler aims to mitigate the number and frequency of compensation code

Trace Scheduling

- Goal is to construct maximal-length acyclic paths through a CFG
  - Applies the list scheduling algorithm to those paths or traces
  - Trace is an acyclic path through the CFG
- Compiler aims to schedule hot paths before cold paths
  - Requires access to profile information

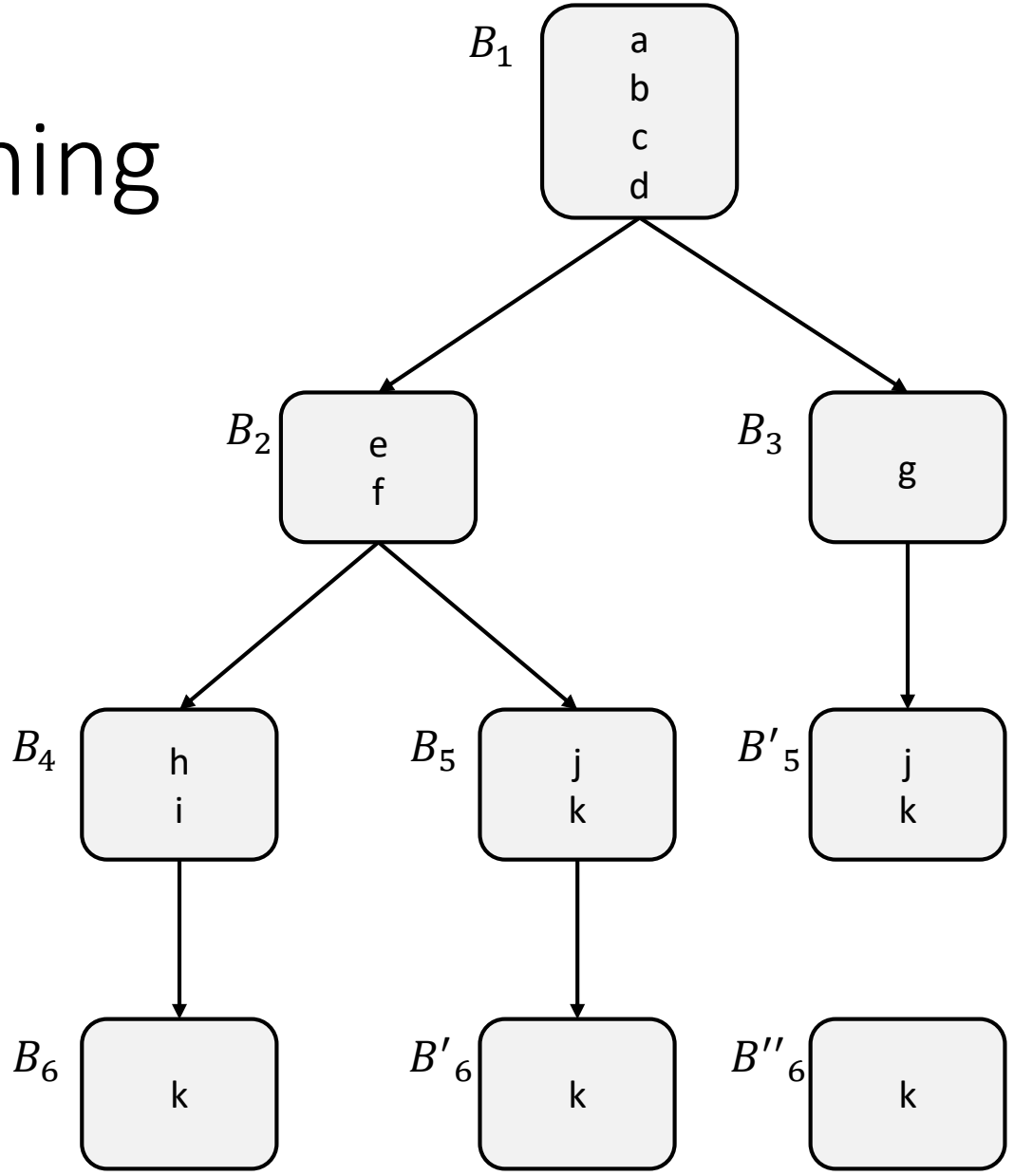


Trace Scheduling

- Selecting edges to form a trace can be greedy
  - For example, a possible trace is $\{B_1, B_2, B_4, B_6\}$
  - Trace construction stops when it runs of edges or there is a loop-closing branch
- Scheduler applies the list scheduling algorithm to traces
  - Schedules a trace, and moves on to the next most-frequently executed trace
- Note there may be entry points in the middle of a trace
  - Blocks may have multiple predecessors
  - Compilers have to be careful performing code motion across such blocks

Scheduling with BB Cloning

- Join or merge BBs limit extending EBB or trace scheduling
- Cloning BBs allows creating longer join-free paths
- After cloning, the entire graph on the right is an EBB
  - Schedule $\{B_1, B_2, B_4, B_6\}$ if hot (say)
  - Then, can schedule either $\{B_5, B'_6\}$ or $\{B_3, B'_5, B''_6\}$



References

- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 12.