

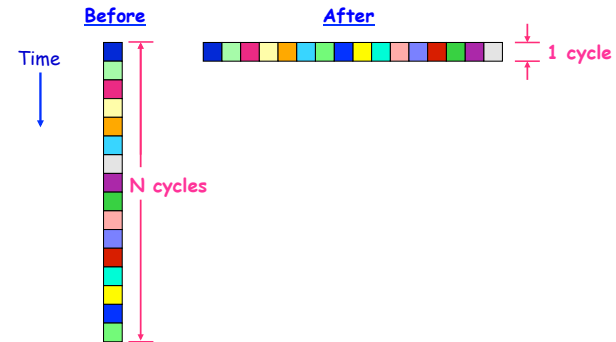
# Lecture 18

## List Scheduling & Global Scheduling

Reading: Chapter 10.3-10.4

### Review: The Ideal Scheduling Outcome

- What prevents us from achieving this ideal?



### Review: Scheduling Constraints

- **Hardware Resources**
  - finite set of FUs with instruction type, bandwidth, and latency constraints
  - cache hierarchy also has many constraints
- **Data Dependences**
  - can't consume a result before it is produced
  - ambiguous dependences create many challenges
- **Control Dependences**
  - impractical to schedule for all possible paths
  - choosing an "expected" path may be difficult
    - recovery costs can be non-trivial if you are wrong

### List Scheduling

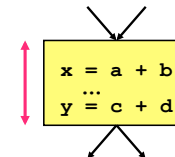
- The most common technique for scheduling instructions **within a basic block**

We don't need to worry about:

- control flow

We do need to worry about:

- data dependences
- hardware resources



- Even without control flow, the problem is still **NP-hard**

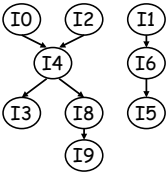
## List Scheduling Algorithm: Inputs and Outputs

Algorithm reproduced from:

- "An Experimental Evaluation of List Scheduling", Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Rice University, Department of Computer Science Technical Report 98-326, September 1998.

### Inputs:

Data Precedence Graph (DPG)



Machine Parameters

# of FUs:  
2 INT, 1 FP  
Latencies:  
add = 1 cycle, ...  
Pipelining:  
1 add/cycle, ...

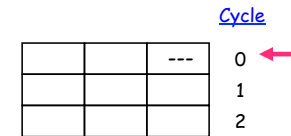
### Output:

Scheduled Code    Cycle

Scheduled Code	Cycle
I0   I2   ---	0
---   I1   I4	1
I3   I8   I6	2
I10   ---   I11	3
I7   I9   I5	4

## List Scheduling: The Basic Idea

- Maintain a list of instructions that are ready to execute
  - data dependence constraints would be preserved
  - machine resources are available
- Moving cycle-by-cycle through the schedule template:
  - choose instructions from the list & schedule them
  - update the list for the next cycle



## What Makes Life Interesting: Choice

Easy case:

- all ready instructions can be scheduled this cycle



Interesting case:

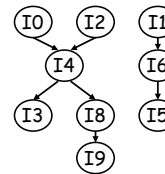
- we need to pick a subset of the ready instructions



- List scheduling makes choices based upon priorities
  - assigning priorities correctly is a key challenge

## Intuition Behind Priorities

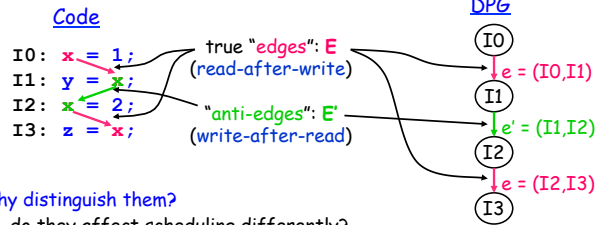
- Intuitively, what should the priority correspond to?
- What factors are used to compute it?
  - data dependences?
  - machine parameters?



# of FUs:  
2 INT, 1 FP  
Latencies:  
add = 1 cycle, ...  
Pipelining:  
1 add/cycle, ...

## Representing Data Dependences: The Data Precedence Graph (DPG)

- Two different kinds of edges:

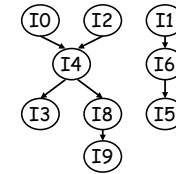


- Why distinguish them?
  - do they affect scheduling differently?
- What about output dependences?

## Computing Priorities

- Let's start with just **true dependences** (i.e. "edges" in DPG)
- Priority = **latency-weighted depth** in the DPG

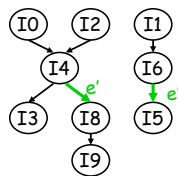
$$priority(x) = \max(\forall l \in leaves(DPG) \forall p \in paths(x, \dots, l) \sum_{p_i=x}^l latency(p_i))$$



## Computing Priorities (Cont.)

- Now let's also take **anti-dependences** into account
  - i.e. anti-edges in the set  $E'$

$$priority(x) = \begin{cases} latency(x) & \text{if } x \text{ is a leaf} \\ \max(latency(x) + \max_{(x,y) \in E}(priority(y)), \max_{(x,y) \in E'}(priority(y))) & \text{otherwise.} \end{cases}$$



## List Scheduling Algorithm

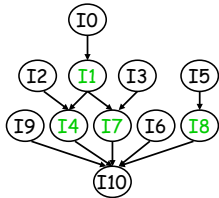
```

cycle = 0;
ready-list = root nodes in DPG; inflight-list = {};

while ((|ready-list|+|inflight-list| > 0) && an issue slot is available) {
  for op = (all nodes in ready-list in descending priority order) {
    if (an FU exists for op to start at cycle) {
      remove op from ready-list and add to inflight-list;
      add op to schedule at time cycle;
      if (op has an outgoing anti-edge)
        add all targets of op's anti-edges that are ready to ready-list;
    }
  }
  cycle = cycle + 1;
  for op = (all nodes in inflight-list)
    if (op finishes at time cycle) {
      remove op from inflight-list;
      check nodes waiting for op & add to ready-list if all operands
      available;
    }
}
    
```

### Example

I0: a = 1  
 I1: f = a + x  
 I2: b = 7  
 I3: c = 9  
 I4: g = f + b  
 I5: d = 13  
 I6: e = 19;  
 I7: h = f + c  
 I8: j = d + y  
 I9: z = -1  
 I10: JMP L1

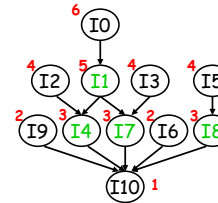


		Cycle
		0
		1
		2
		3
		4
		5
		6

- 2 identical fully-pipelined FUs
- adds take 2 cycles; all other insts take 1 cycle

### What if We Break Ties Differently?

I0: a = 1  
 I1: f = a + x  
 I2: b = 7  
 I3: c = 9  
 I4: g = f + b  
 I5: d = 13  
 I6: e = 19;  
 I7: h = f + c  
 I8: j = d + y  
 I9: z = -1  
 I10: JMP L1

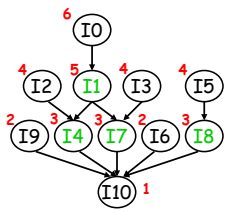


		Cycle
		0
		1
		2
		3
		4
		5
		6

- 2 identical fully-pipelined FUs
- adds take 2 cycles; all other insts take 1 cycle

### Contrasting the Two Schedules

- Breaking ties arbitrarily may not be the best approach



		Cycle
I0	I2	0
I1	I3	1
I5	I9	2
I4	I7	3
I8	I6	4
---	---	5
I10		6

		Cycle
I0	I2	0
I1	I5	1
I3	I8	2
I4	I7	3
I9	I6	4
I10		5

### Backward List Scheduling

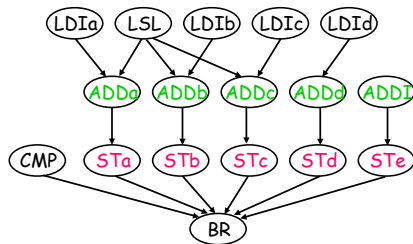
Modify the algorithm as follows:

- reverse the direction of all edges in the DPG
- schedule the finish times of each operation
  - start times must still be used to ensure FU availability

Impact of scheduling backwards:

- clusters operations near the end (vs. the beginning)
- may be either better or worse than forward scheduling

### Backward List Scheduling Example: Let's Schedule it Forward First

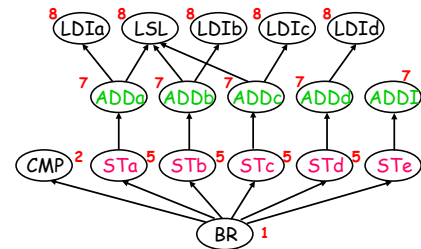


INT	INT	MEM	Cycle
LDIa	LSL	----	0
LDIb	LDIc	----	1
LDId	ADDa	----	2
ADDb	ADDc	----	3
ADDd	ADDI	STa	4
CMP	----	STb	5
----	----	STc	6
----	----	STd	7
----	----	STe	8
----	----	----	9
----	----	----	10
----	----	----	11
BR	----	----	12

Hardware parameters:

- 2 INT units: ADDs take 2 cycles; others take 1 cycle
- 1 MEM unit: stores (ST) take 4 cycles

### Now Let's Try Scheduling Backward



INT	INT	MEM	Cycle
LDIa	----	----	0
ADDI	LSL	----	1
ADDd	LDIc	----	2
ADDc	LDId	STe	3
ADDb	LDIa	STd	4
ADDa	----	STc	5
----	----	STb	6
----	----	STa	7
----	----	----	8
----	----	----	9
CMP	----	----	10
BR	----	----	11

Hardware parameters:

- 2 INT units: ADDs take 2 cycles; others take 1 cycle
- 1 MEM unit: stores (ST) take 4 cycles

### Contrasting Forward vs. Backward List Scheduling

Forward				Backward			
INT	INT	MEM	Cycle	INT	INT	MEM	Cycle
LDIa	LSL	----	0	LDIa	----	----	0
LDIb	LDIc	----	1	ADDI	LSL	----	1
LDId	ADDa	----	2	ADDd	LDIc	----	2
ADDb	ADDc	----	3	ADDc	LDId	STe	3
ADDd	ADDI	STa	4	ADDb	LDIa	STd	4
CMP	----	STb	5	ADDa	----	STc	5
----	----	STc	6	----	----	STb	6
----	----	STd	7	----	----	STa	7
----	----	STe	8	----	----	----	8
----	----	----	9	----	----	----	9
----	----	----	10	----	----	----	10
----	----	----	11	CMP	----	----	10
BR	----	----	12	BR	----	----	11

- backward scheduling clusters work near the end
- backward is better in this case, but this is not always true

### Evaluation of List Scheduling

Cooper et al. propose "RBF" scheduling:

- schedule each block *M* times forward & backward
- break any priority ties randomly

For real programs:

- regular list scheduling works very well

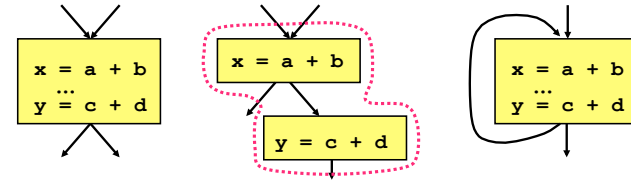
For synthetic blocks:

- RBF wins when "available parallelism" (AP) is ~2.5
- for smaller AP, scheduling is too constrained
- for larger AP, any decision tends to work well

### List Scheduling Wrap-Up

- The **priority** function can be **arbitrarily sophisticated**
  - e.g., filling branch delay slots in early RISC processors
- List scheduling is widely used, and it works fairly well
- It is limited, however, by basic block boundaries

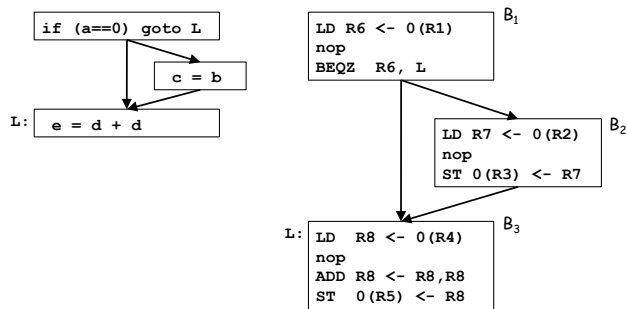
### Scheduling Roadmap



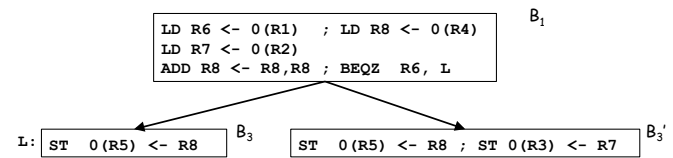
- List Scheduling:** • within a basic block
- Global Scheduling:** • *across* basic blocks
- Software Pipelining:** • *across* loop iterations

### Introduction to Global Scheduling

Assume each clock can execute 2 operations of any kind.



### Result of Code Scheduling



## Terminology

### Control equivalence:

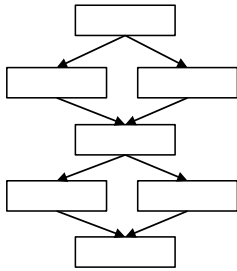
- Two operations  $o_1$  and  $o_2$  are *control equivalent* if  $o_1$  is executed if and only if  $o_2$  is executed.

### Control dependence:

- An op  $o_2$  is *control dependent* on op  $o_1$  if the execution of  $o_2$  depends on the outcome of  $o_1$ .

### Speculation:

- An operation  $o$  is *speculatively* executed if it is executed before all the operations it depends on (control-wise) have been executed.
- Requirements:**
  - does not raise an exception
  - satisfies data dependences



## Code Motions

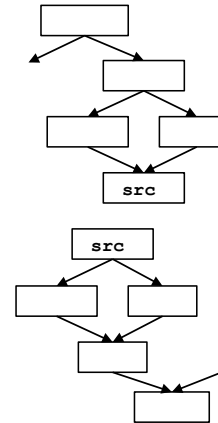
**Goal:** Shorten execution time *probabilistically*

### Moving instructions up:

- Move instruction to a cut set (from entry)
- Speculation:** even when not anticipated.

### Moving instructions down:

- Move instruction to a cut set (from exit)
- May execute extra instruction
- Can duplicate code



## General-Purpose Applications

- Lots of data dependences
- Key performance factor: **memory latencies**
- Move memory fetches up**
  - Speculative memory fetches can be expensive
- Control-intensive: get execution profile**
  - Static estimation**
    - Innermost loops are frequently executed
      - back edges are likely to be taken
    - Edges that branch to exit and exception routines are not likely to be taken
  - Dynamic profiling**
    - Instrument code and **measure** using representative data

## A Basic Global Scheduling Algorithm

- Schedule innermost loops first
- Only upward code motion
- No creation of copies
- Only one level of speculation

## Program Representation

- A **region** in a control flow graph is:
  - a set of **basic blocks** and all the **edges** connecting these blocks,
  - such that control from outside the region **must enter through a single entry block**.
- A **procedure** is represented as a **hierarchy of regions**
  - The whole control flow graph is a region
  - Each natural loop in the flow graph is a region
  - Natural loops are hierarchically nested
- **Schedule regions from inner to outer**
  - treat inner loop as a black box unit
    - can **schedule around it but not into it**
  - **ignore all the loop back edges** → get an acyclic graph

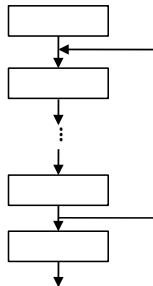
## Algorithm

```
Compute data dependencies;  
For each region from inner to outer {  
  For each basic block B in prioritized topological order {  
    CandBlocks = ControlEquiv{B} U  
                 Dominated-Successors{ControlEquiv{B}};  
    CandInsts = ready operations in CandBlocks;  
    For (t = 0, 1, ... until all operations from B are scheduled) {  
      For (n in CandInst in priority order) {  
        if (n has no resource conflicts at time t) {  
          S(n) = < B, t >  
          Update resource commitments  
          Update data dependencies  
        }  
      }  
      Update CandInsts;  
    }  
  }  
}
```

**Priority functions:** non-speculative before speculative

## Extensions

- **Prepass before scheduling: loop unrolling**
- **Especially important to move operation up loop back edges**



## Summary

- **Global scheduling**
  - Legal code motions
  - Heuristics