Introduction to Instruction Scheduling

EaC Ch. 12

Slides updated by Christophe Dubach, Winter 2024

Copyright 2010, Keith D. Cooper & 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.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

What Makes Code Run Fast?

- Many operations have non-zero latencies
- Modern machines can issue several operations per cycle
- Execution time is *order-dependent*
  (and has been since the 60's)

Assumed latencies *(conservative)*

Operation	Cycles
load/loadAI	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- Loads & stores may or may not block
  > Non-blocking ⇒fill those issue slots
- Branch costs vary with path taken
- Branches typically have delay slots
  > Fill slots with unrelated operations
  > Percolates branch upward
- Scheduler should hide the latencies

Example

w ← w * 2 * x * y * z

Simple schedule

```
1    loadAI   r0, @w    => r1
4    add      r1, r1    => r1
5    loadAI   r0, @x    => r2
8    mult     r1, r2    => r1
9    loadAI   r0, @y    => r2
12   mult     r1, r2    => r1
13   loadAI   r0, @z    => r2
16   mult     r1, r2    => r1
18   storeAI  r1, @w    => r0
21   r1 is free
```

2 registers, 20 cycles

Schedule loads early

```
1    loadAI   r0, @w    => r1
2    loadAI   r0, @x    => r2
3    loadAI   r0, @y    => r3
4    add      r1, r1    => r1
5    mult     r1, r2    => r1
6    loadAI   r0, @z    => r2
7    mult     r1, r3    => r1
9    mult     r1, r2    => r1
11   storeAI  r1, @w    => r0
14   r1 is free
```

3 registers, 13 cycles

Reordering operations for speed is called instruction scheduling

ALU Characteristics

This data is surprisingly hard to measure accurately

- Value-dependent behavior
- Context-dependent behavior
- Compiler behavior
- Difficult to reconcile measurement with the data in the manuals

Intel Xeon E5530 (Mar. 2009) operation latencies

Instruction	Cost
64 bit integer subtract	1
64 bit integer multiply	3
64 bit integer divide	41
Double precision add	3
Double precision subtract	3
Double precision multiply	5
Double precision divide	22
Single precision add	3
Single precision subtract	3
Single precision multiply	4
Single precision divide	14

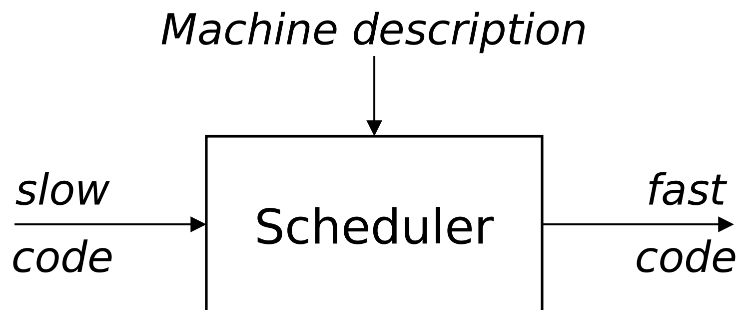
PACE
platform-aware compilation environment

Instruction Scheduling *(Engineer's View)*

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

The Concept



The task

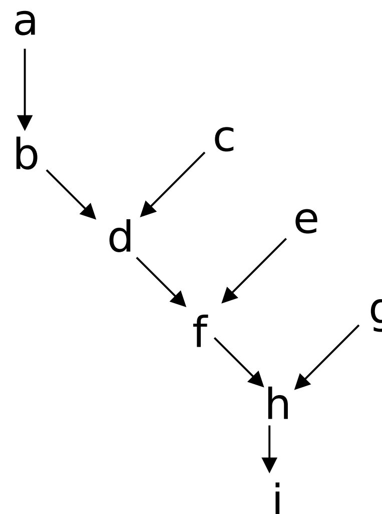
- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently

To capture properties of the code, build a precedence graph *G*

- Nodes $n \in G$ are operations with *type(n)* and *delay(n)*
- An edge $e = (n_1,n_2) \in G$ if & only if n_2 uses the result of n_1

```
a:  loadAI   r0, @w   => r1
b:  add      r1, r1   => r1
c:  loadAI   r0, @x   => r2
d:  mult     r1, r2   => r1
e:  loadAI   r0, @y   => r2
f:  mult     r1, r2   => r1
g:  loadAI   r0, @z   => r2
h:  mult     r1, r2   => r1
i:  storeAI  r1, @w   => r0
```

The Code



The Precedence Graph

Instruction Scheduling *(Definitions)*

A correct schedule S maps each $n \in N$ into a non-negative integer representing its cycle number, and

1. $S(n) \geq 0$, for all $n \in N$ (obviously)

2. If $(n_1, n_2) \in E$, $S(n_1) + delay(n_1) \leq S(n_2)$

3. For each type *t*, there are no more operations of type *t* in any cycle than the target machine can issue

The length of a schedule *S*, denoted $L(S)$, is

$$L(S) = max_{n \in N} (S(n) + delay(n))$$

The goal is to find the shortest possible correct schedule.

S_{opt} is time-optimal if $L(S_{opt}) \leq L(S_i)$, for all other schedules S_i

A schedule might also be optimal in terms of registers, power, or space….

Instruction Scheduling *(What's so difficult?)*

Critical Points

- All operands must be available
- Multiple operations can be *ready*
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

Together, these issues make scheduling *hard* (NP-Complete)

Local scheduling is the simple case

- Restricted to straight-line code
- Assumes consistent and predictable latencies

Instruction Scheduling: The big picture

1. Build a precedence graph, *P*
2. Compute a *priority function* over the nodes in *P*
3. Use list scheduling to construct a schedule, 1 cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose a ready operation and schedule it
      II. Update the ready queue

Local list scheduling

- The dominant algorithm for twenty+ years
- A greedy, heuristic, local technique (within a basic block)

Local List Scheduling

```
Cycle ← 1
Ready ← leaves of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
  if (Ready ≠ Ø) then
    remove highest priority op from Ready
    S(op) ← Cycle
    Active ← Active ∪ op

  Cycle ← Cycle + 1

  for each op ∈ Active
    if (S(op) + delay(op) ≤ Cycle) then
      remove op from Active
      for each successor s of op in P
        if (s is ready) then
          Ready ← Ready ∪ s
```

Removal in priority order

op has completed execution

If successor's operands are ready, add it to **Ready**

Scheduling Example

1. Build the precedence graph

```
a:  loadAI   r0, @w   => r1
b:  add      r1, r1   => r1
c:  loadAI   r0, @x   => r2
d:  mult     r1, r2   => r1
e:  loadAI   r0, @y   => r2
f:  mult     r1, r2   => r1
g:  loadAI   r0, @z   => r2
h:  mult     r1, r2   => r1
i:  storeAI  r1, @w   => r0
```

The Code

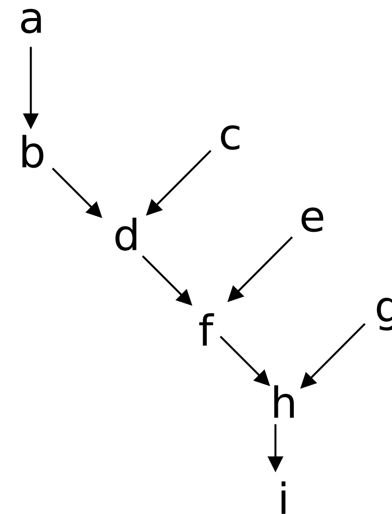
The Precedence Graph

Scheduling Example

1. Build the precedence graph

```
a:  loadAI   r0, @w   => r1
b:  add      r1, r1   => r1
c:  loadAI   r0, @x   => r2
d:  mult     r1, r2   => r1
e:  loadAI   r0, @y   => r2
f:  mult     r1, r2   => r1
g:  loadAI   r0, @z   => r2
h:  mult     r1, r2   => r1
i:  storeAI  r1, @w   => r0
```

The Code



The Precedence Graph

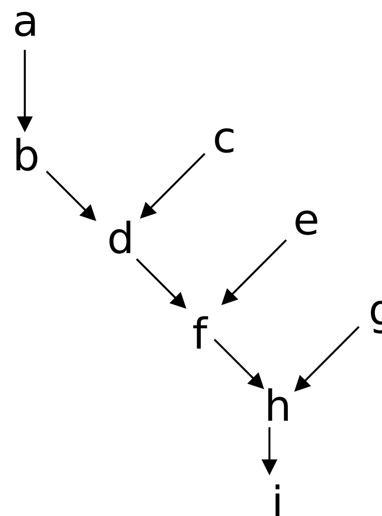
Scheduling Example

Operation	Cycles
load/loadAI	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0, @w   => r1
b:  add      r1, r1   => r1
c:  loadAI   r0, @x   => r2
d:  mult     r1, r2   => r1
e:  loadAI   r0, @y   => r2
f:  mult     r1, r2   => r1
g:  loadAI   r0, @z   => r2
h:  mult     r1, r2   => r1
i:  storeAI  r1, @w   => r0
```

The Code



The Precedence Graph

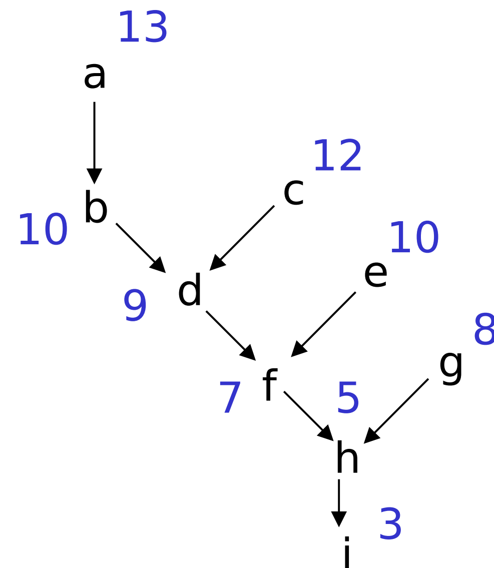
Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path

Operation	Cycles
load/loadAI	3
store	3
loadI	1
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

```
a:  loadAI   r0, @w   => r1
b:  add      r1, r1   => r1
c:  loadAI   r0, @x   => r2
d:  mult     r1, r2   => r1
e:  loadAI   r0, @y   => r2
f:  mult     r1, r2   => r1
g:  loadAI   r0, @z   => r2
h:  mult     r1, r2   => r1
i:  storeAI  r1, @w   => r0
```

The Code



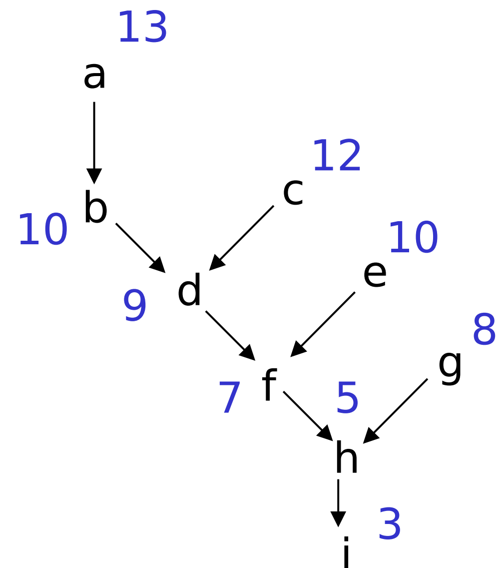
The Precedence Graph

Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

```
a:  loadAI   r0, @w   => r1
b:  add      r1, r1   => r1
c:  loadAI   r0, @x   => r2
d:  mult     r1, r2   => r1
e:  loadAI   r0, @y   => r2
f:  mult     r1, r2   => r1
g:  loadAI   r0, @z   => r2
h:  mult     r1, r2   => r1
i:  storeAI  r1, @w   => r0
```

The Code



The Precedence Graph

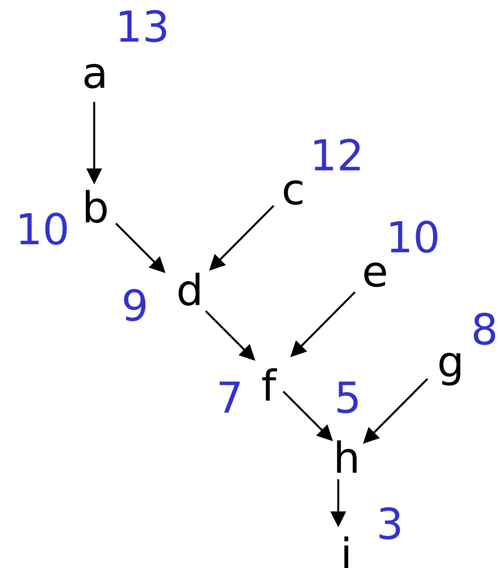
Scheduling Example

1. Build the precedence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

New register name used

```
1   loadAI   r0, @w   => r1
2   loadAI   r0, @x   => r2
3   loadAI   r0, @y   => r3
4   add      r1, r1   => r1
5   mult     r1, r2   => r1
6   loadAI   r0, @z   => r2
7   mult     r1, r3   => r1
9   mult     r1, r2   => r1
11  storeAI  r1, @w   => r0
```

Scheduled Code



The Precedence Graph

More on List Scheduling

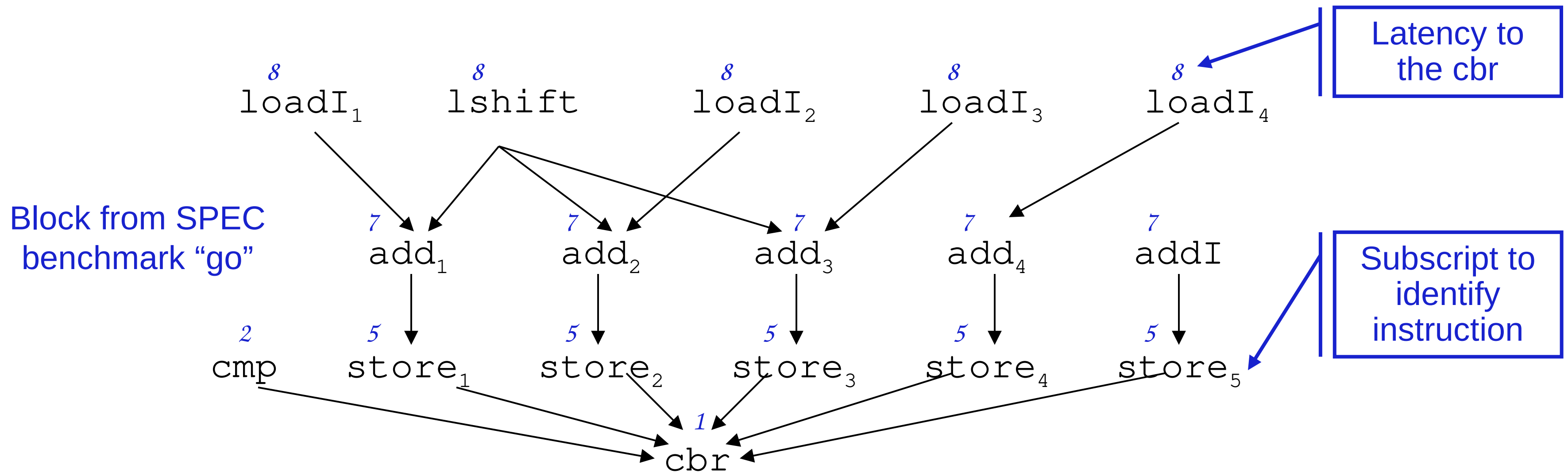
List scheduling breaks down into two distinct classes

Forward list scheduling	Backward list scheduling
• Start with available operations	• Start with no successors
• Work forward in time	• Work backward in time
• Ready ⇒ all operands available	• Ready ⇒ latency covers uses

Variations on list scheduling

- Prioritize critical path(s)
- Schedule last use as soon as possible
- Depth first in precedence graph (minimize registers)
- Breadth first in precedence graph (minimize interlocks)
- Prefer operation with most successors

Local Scheduling



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

- Assuming the machine can execute at each cycle:
  - 2 ALU operations (including loadI, cmp, branch)
  - 1 memory operation (e.g. store or load)

Local Scheduling (using latency to root as priority)

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

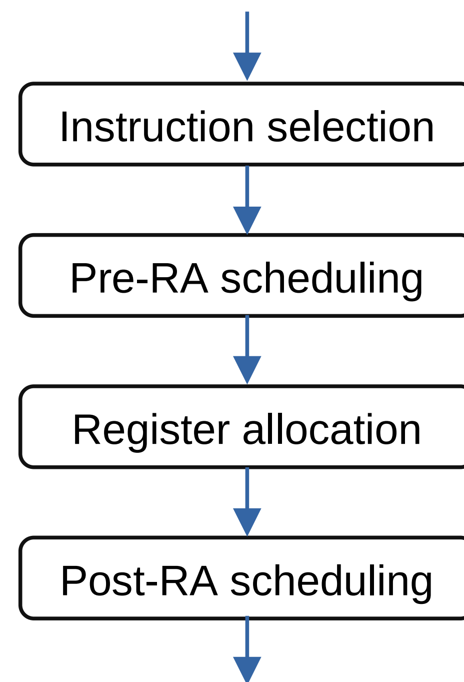
	Int	Int	Mem
1	$loadI_4$		
2	addI	lshift	
3	add_4	$loadI_3$	
4	add_3	$loadI_2$	$store_5$
5	add_2	$loadI_1$	$store_4$
6	add_1		$store_3$
7			$store_2$
8			$store_1$
9			
10			
11	cmp		
12	cbr		

Forward and backward can produce different results

The more complete picture

Exemple: LLVM compilation flow

- Instruction selection
  - choose best instructions that matches IR
- Pre-RA instruction scheduling
  - performed on virtual register
  - tries to minimize register pressure
- Register Allocation (RA)
  - introduce physical registers
  - goal is to minimize spilling
- Post-RA instruction scheduling
  - help scheduling spill code
  - more constrained (physical registers introduce false dependencies and cannot introduce new registers)



Next Lecture

- Object Oriented Programming Support