



COMP 412
FALL 2009

Local Instruction Scheduling
— A Primer for Lab 3 —
Comp 412

Copyright 2009, 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)

<u>Operation</u>	<u>Cycles</u>
load	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 \Rightarrow 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

Lab 3 will build a local scheduler

Example



$$w \leftarrow w * 2 * x * y * z$$

Simple schedule

1	loadAl	r0,@w	⇒ r1
4	add	r1,r1	⇒ r1
5	loadAl	r0,@x	⇒ r2
8	mult	r1,r2	⇒ r1
9	loadAl	r0,@y	⇒ r2
12	mult	r1,r2	⇒ r1
13	loadAl	r0,@z	⇒ r2
16	mult	r1,r2	⇒ r1
18	storeAl	r1	⇒ r0,@w
21	r1 is free		

2 registers, 20 cycles

Schedule loads early

1	loadAl	r0,@w	⇒ r1
2	loadAl	r0,@x	⇒ r2
3	loadAl	r0,@y	⇒ r3
4	add	r1,r1	⇒ r1
5	mult	r1,r2	⇒ r1
6	loadAl	r0,@z	⇒ r2
7	mult	r1,r3	⇒ r1
9	mult	r1,r2	⇒ r1
11	storeAl	r1	⇒ r0,@w
14	r1 is free		

3 registers, 13 cycles

Reordering operations for speed is called *instruction scheduling*

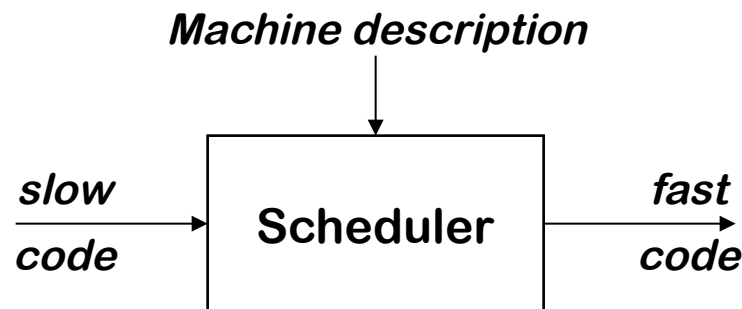
Instruction Scheduling (Engineer's View)



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

Instruction Scheduling (The Abstract View)

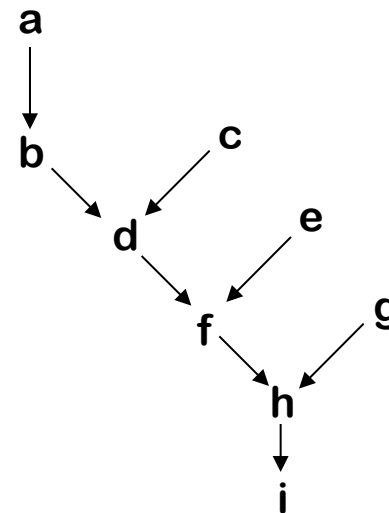


To capture properties of the code, build a precedence graph G

- 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:	loadAl	r0,@w	\Rightarrow	r1
b:	add	r1,r1	\Rightarrow	r1
c:	loadAl	r0,@x	\Rightarrow	r2
d:	mult	r1,r2	\Rightarrow	r1
e:	loadAl	r0,@y	\Rightarrow	r2
f:	mult	r1,r2	\Rightarrow	r1
g:	loadAl	r0,@z	\Rightarrow	r2
h:	mult	r1,r2	\Rightarrow	r1
i:	storeAl	r1	\Rightarrow	r0,@w

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 $(n_1, n_2) \in E$, $S(n_1) + \text{delay}(n_1) \leq S(n_2)$
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

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

The goal is to find the shortest possible correct schedule.

S is time-optimal if $L(S) \leq L(S_1)$, for all other schedules S_1

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
- 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 the highest priority ready operation & schedule it
 - II. Update the ready queue

Local list scheduling

- The dominant algorithm for thirty years
- A greedy, heuristic, local technique

Editorial: My one complaint about the literature on scheduling is that it does not provide us with an analogy that gives us leverage on the problem in the way that coloring does with allocation.

List scheduling is just an algorithm. It doesn't give us intuitions about the problem.



Local List Scheduling

```
Cycle  $\leftarrow$  1
Ready  $\leftarrow$  leaves of  $P$ 
Active  $\leftarrow$   $\emptyset$ 

while (Ready  $\cup$  Active  $\neq$   $\emptyset$ )
  if (Ready  $\neq$   $\emptyset$ ) then
    remove an  $op$  from Ready
     $S(op) \leftarrow$  Cycle
    Active  $\leftarrow$  Active  $\cup$   $op$ 

  Cycle  $\leftarrow$  Cycle + 1

  for each  $op \in$  Active
    if ( $S(op) + \text{delay}(op) \leq$  Cycle) then
      remove  $op$  from Active
      for each successor  $s$  of  $op$  in  $P$ 
        if ( $s$  is ready) then
          Ready  $\leftarrow$  Ready  $\cup$   $s$ 
```

Removal in priority order

op has completed execution

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

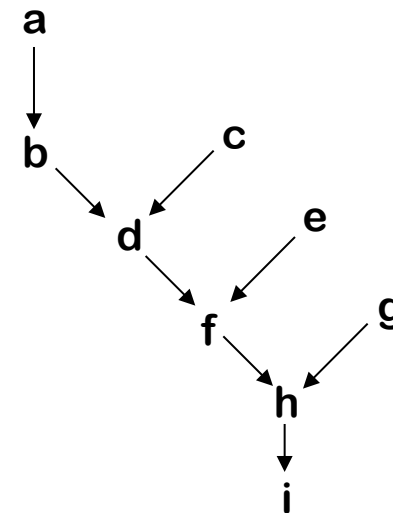


Scheduling Example

1. Build the precedence graph

a:	loadAl	r0,@w	⇒ r1
b:	add	r1,r1	⇒ r1
c:	loadAl	r0,@x	⇒ r2
d:	mult	r1,r2	⇒ r1
e:	loadAl	r0,@y	⇒ r2
f:	mult	r1,r2	⇒ r1
g:	loadAl	r0,@z	⇒ r2
h:	mult	r1,r2	⇒ r1
i:	storeAl	r1	⇒ r0,@w

The Code



The Precedence Graph

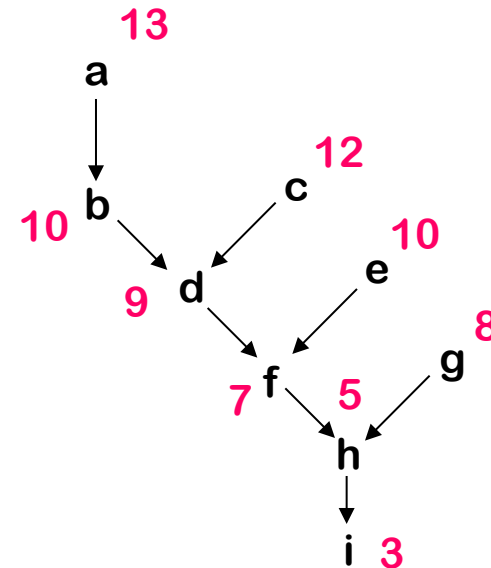


Scheduling Example

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

a:	loadAl	r0,@w	⇒ r1
b:	add	r1,r1	⇒ r1
c:	loadAl	r0,@x	⇒ r2
d:	mult	r1,r2	⇒ r1
e:	loadAl	r0,@y	⇒ r2
f:	mult	r1,r2	⇒ r1
g:	loadAl	r0,@z	⇒ r2
h:	mult	r1,r2	⇒ r1
i:	storeAl	r1	⇒ r0,@w

The Code



The Precedence Graph

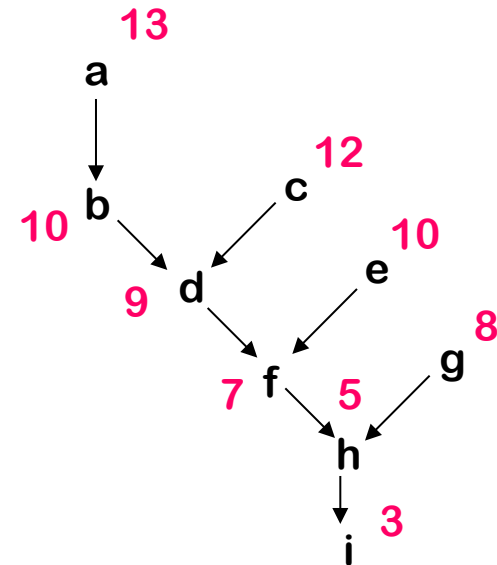


Scheduling Example

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

1) a:	loadAl	r0,@w	⇒ r1
2) c:	loadAl	r0,@x	⇒ r2
3) e:	loadAl	r0,@y	⇒ r3
4) b:	add	r1,r1	⇒ r1
5) d:	mult	r1,r2	⇒ r1
6) g:	loadAl	r0,@z	⇒ r2
7) f:	mult	r1,r3	⇒ r1
9) h:	mult	r1,r2	⇒ r1
11) i:	storeAl	r1	⇒ r0,@w

Used a new register name



The Code

The Precedence Graph

Improving the Efficiency of Scheduling



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

while (Ready ∪ Active ≠ ∅)
  if (Ready ≠ ∅) then
    remove an 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
```

Updating the Ready queue is the most expensive part of this algorithm

- Can create a separate queue for each cycle
 - Add op to queue for cycle when it will complete
 - Naive implementation needs too many queues
- Can use fewer queues
 - Number of queues bounded by maximum operation latency
 - Use them in a cyclic (or modulo) fashion
- Detailed algorithm is in the next couple of slides



Detailed Scheduling Algorithm I

Idea: Keep a collection of worklists $W[c]$, one per cycle

- We need $\text{MaxC} = \text{max delay} + 1$ such worklists
- Precedence graph is (N,E)

Code:

```
for each  $n \in N$  do begin count[n] := 0; earliest[n] = 0 end  
for each  $(n1,n2) \in E$  do begin  
    count[n2] := count[n2] + 1;  
    successors[n1] := successors[n1]  $\cup$  {n2};  
end  
for  $i := 0$  to  $\text{MaxC} - 1$  do  $W[i] := \emptyset$ ;  
Wcount := 0;  
for each  $n \in N$  do  
    if count[n] = 0 then begin  
         $W[0] := W[0] \cup \{n\}$ ; Wcount := Wcount + 1;  
    end  
 $c := 0$ ; // c is the cycle number  
 $cW := 0$ ; // cW is the number of the worklist for cycle c  
instr[c] :=  $\emptyset$ ;
```



Detailed Scheduling Algorithm II

```
while Wcount > 0 do begin
  while W[cW] =  $\emptyset$  do begin
    c := c + 1; instr[c] :=  $\emptyset$ ; cW := mod(cW+1,MaxC);
  end
  nextc := mod(c+1,MaxC);
  while W[cW]  $\neq$   $\emptyset$  do begin
Priority  $\longrightarrow$  select and remove an arbitrary instruction x from W[cW];
    if  $\exists$  free issue units of type(x) on cycle c then begin
      instr[c] := instr[c]  $\cup$  {x}; Wcount := Wcount - 1;
      for each y  $\in$  successors[x] do begin
        count[y] := count[y] - 1;
        earliest[y] := max(earliest[y], c+delay(x));
        if count[y] = 0 then begin
          loc := mod(earliest[y],MaxC);
          W[loc] := W[loc]  $\cup$  {y}; Wcount := Wcount + 1;
        end
      end
    else W[nextc] := W[nextc]  $\cup$  {x};
  end
end
```



More List Scheduling

List scheduling breaks down into two distinct classes

Forward list scheduling

- Start with available operations
- Work forward in time
- Ready \Rightarrow all operands available

Backward list scheduling

- Start with no successors
- Work backward in time
- Ready \Rightarrow 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



My Favorite List Scheduling Variant

Schielke's RBF Algorithm

- Recognize that forward & backward both have their place
- Recognize that tie-breaking matters & that, while we can rationalize various tie-breakers, we do not understand them
 - Part of approximating the solution to an NP-complete problem

The Algorithm

- Run forward scheduler 5 times, breaking ties randomly
- Run backward scheduler 5 time, breaking ties randomly
- Keep the best schedule

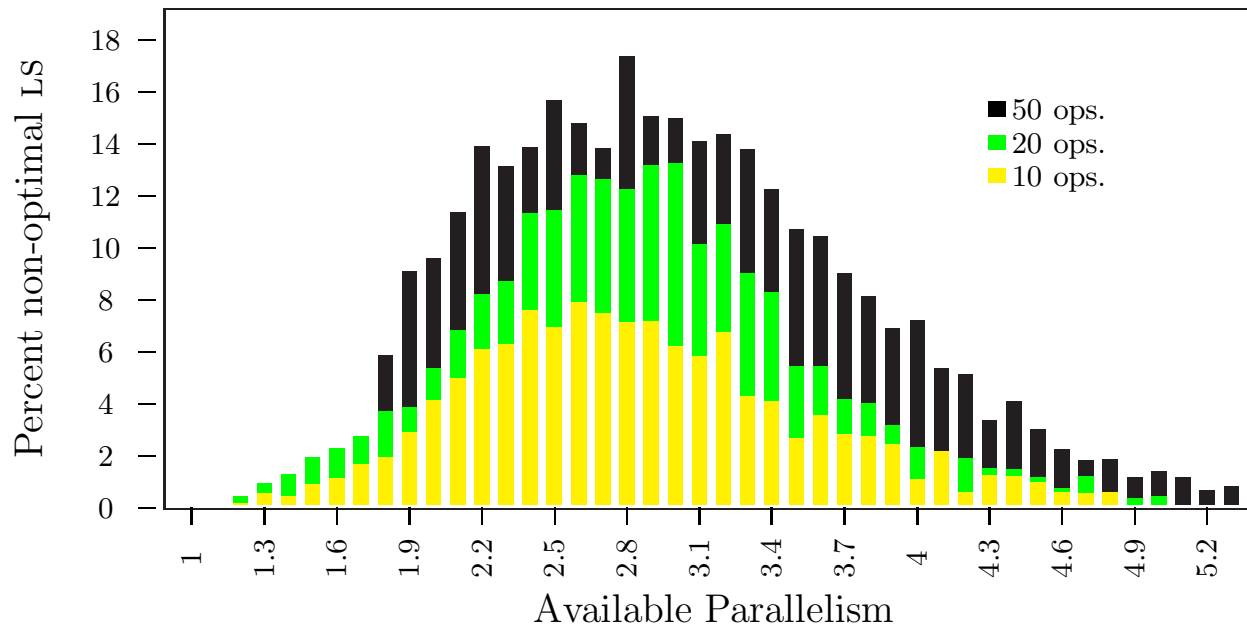
RBF means "randomized backward & forward"

3 compute months of work



How Good is List Scheduling? Schielke's Study

Non-optimal list schedules (%) versus available parallelism
1 functional unit, randomly generated blocks of 10, 20, 50 ops



- 85,000 randomly generated blocks
- RBF found optimal schedules for > 80%
- Peak difficulty (for RBF) is around 2.8

Tie-breaking matters because it affects the choice when queue has > 1 element

Lab 3



- Implement two schedulers for basic blocks in ILOC
- One must be list scheduling with specified priority
- Other can be a second priority, or a different algorithm
- Same ILOC subset as in lab 1 (plus NOP)
- Different execution model
 - Two asymmetric functional units
 - Latencies different from lab 1
 - Simulator different from lab 1

Lab 3 — Specific questions



1. *Are we in over our heads?* No. The hard part of this lab should be trying to get good code. The programming is not bad; you can reuse some stuff from lab 1. You have all the specifications & tools you need. Jump in and start programming.
2. *How many registers can we use?* Assume that you have as many registers as you need. If the simulator runs out, we'll get more. Rename registers to avoid false dependences & conflicts.
3. *What about a store followed by a load?* If you can **show** that the two operations must refer to different memory locations, the scheduler can overlap their execution. Otherwise, the store must complete before the load issues.
4. *What about test files?* We will put some test files online on OwlNet. We will test your lab on files you do not see. We have had problems (in the past) with people optimizing their labs for the test data. (Not that you would do that!)

Lab 3 - Hints



- Begin by renaming registers to eliminate false dependencies
- Pay attention to loads and stores
 - When can they be reordered?
- Understand the simulator
- Inserting NOPs may be necessary to get correct code
- Tiebreakers can make a difference