RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Instruction Scheduling (part 2)

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

- Recitations and office hours start this week (today!)
- Office hours have been posted soon
- First homework will be posted by Friday
- First project will be instruction scheduling

Local Instruction Scheduling

Readings: EaC 12.1-12.3, Appendix A (ILOC)

Definition

A *basic block* is a maximal length segment of straight-line (*i.e.*, branch free) code. Control can only enter at first instruction of basic block and exit after last instruction.

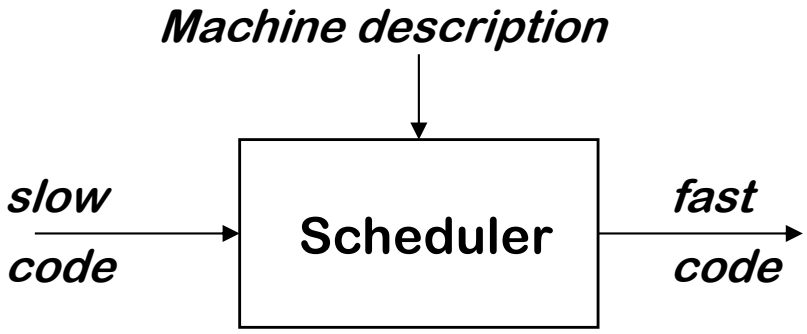
Local: within single basic block
Global: across procedures/functions

Instruction Scheduling (Engineer's View)

The Problem

Given a code fragment (basic block) 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 (idle) cycles**
- **Scheduler operates efficiently**

The Optimization Goal: Generate fast code

Data Dependences (stmt./instr. level)

Dependences ⇒ defined on memory locations / registers

Statement/instruction b depends on statement/instruction a if there exists:

- true of flow dependence
  a writes a location/register that b later reads (RAW conflict)

- anti dependence
  a reads a location/register that b later writes (WAR conflict)

- output dependence
  a writes a location/register that b later writes (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.

true	anti	output
a =	= a	a =
= a	a =	a =

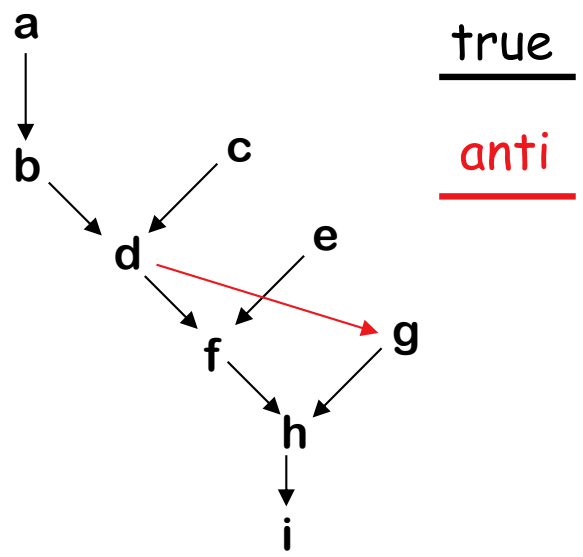
Instruction Scheduling (The Abstract View)

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

- Nodes $n \in G$ are operations with *type(n)* and *delay(n)*
- An edge $e = (n_1, n_2) \in G$ if n_2 depends on 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  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Precedence Graph

All other dependences (output & anti) are covered, i.e., are satisfied through the dependencies shown

Example latencies

Operation	Cycles (latency/delay)
load	3
loadI	1
loadAI	3
store	3
storeAI	3
add	1
mult	2
fadd	1
fmult	2
shift	1
output	1
outputAI	1

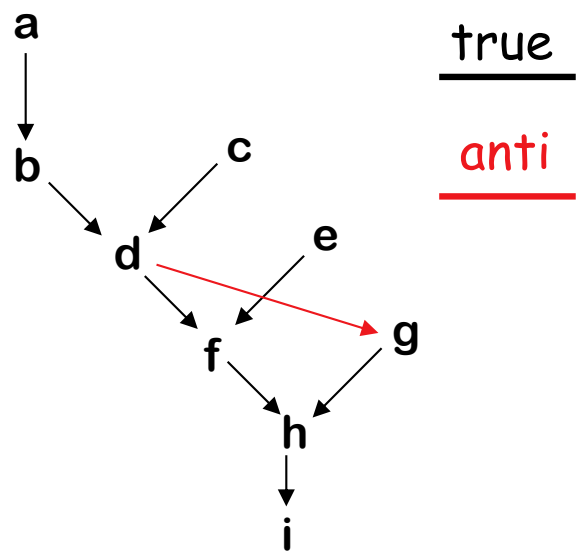
Instruction Scheduling (The Abstract View)

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

- Nodes $n \in G$ are operations with *delay(n)*
- An edge $e = (n_1, n_2) \in G$ if n_2 depends on n_1

```
S(n):
   0  a:  loadAI   r0,@w  ⇒ r1
   3  b:  add      r1,r1  ⇒ r1
   4  c:  loadAI   r0,@x  ⇒ r2
   7  d:  mult     r1,r2  ⇒ r1
   8  e:  loadAI   r0,@y  ⇒ r3
  11  f:  mult     r1,r3  ⇒ r1
  12  g:  loadAI   r0,@z  ⇒ r2
  15  h:  mult     r1,r2  ⇒ r1
  17  i:  storeAI  r1     ⇒ r0,@w
  20
⇒ 20
cycles
```

The Code



The Precedence/Dependence Graph

All other dependences (output & anti) are covered, i.e., are satisfied through the dependencies shown

Instruction Scheduling

The big picture

1. Build a dependence graph, *P*
2. Compute a *priority function* over the nodes in *P*
3. Use list scheduling to construct a schedule, one cycle at a time
   (can only issue/schedule at most one instructions per cycle)
   a. Use a set of operations that are ready
   b. At each cycle
      I. Choose a ready operation (priority-based) and schedule it
      II. Increment cycle
      III. Update the ready set

Local list scheduling

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

Scheduling Example

Operation	Cycles
load	3
loadI	1
loadAI	3
store	3
storeAI	3
add	1
mult	2
fadd	1
fmult	2
shift	1
output	1
outputAI	1

Build a simple local scheduler (basic block)

- non-blocking loads & stores
- out of order execution
- different latencies load/store vs. arith. etc. operations
- different heuristics
- forward / backward scheduling

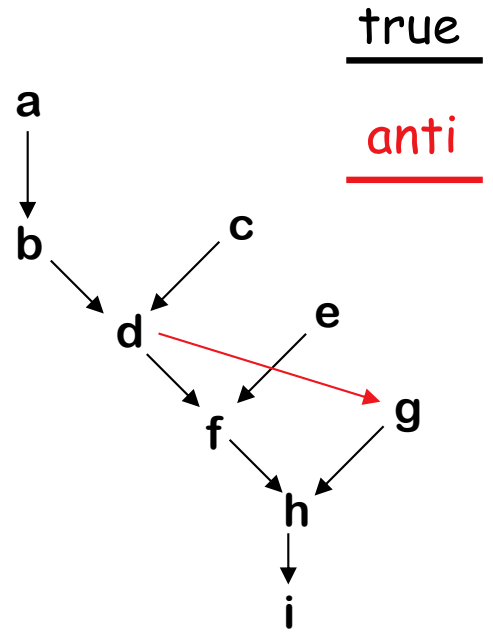
Scheduling Example

1. Build the dependence graph

```
S(n):
    0   a:  loadAI   r0,@w  ⇒ r1
    3   b:  add      r1,r1  ⇒ r1
    4   c:  loadAI   r0,@x  ⇒ r2
    7   d:  mult     r1,r2  ⇒ r1
    8   e:  loadAI   r0,@y  ⇒ r3
   11   f:  mult     r1,r3  ⇒ r1
   12   g:  loadAI   r0,@z  ⇒ r2
   15   h:  mult     r1,r2  ⇒ r1
   17   i:  storeAI  r1     ⇒ r0,@w
   20
```

The Code

⇒ **20 cycles**



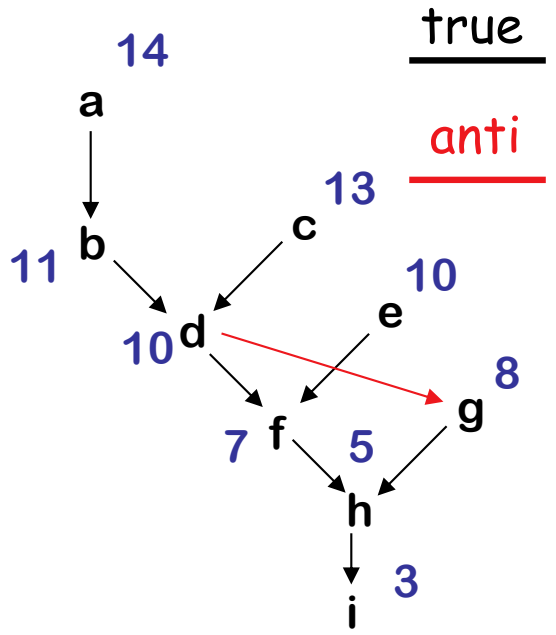
The Dependence Graph

Scheduling Example

1. Build the dependence 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  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

List Scheduling Example

The Code

```
a: loadAI. r0, @w  => r1
b: add      r1, r1 => r1
c: loadAI  r0, @x  => r2
d: mult     r1, r2 => r1
e: loadAI  r0, @y  => r3
f: mult     r1, r3 => r1
g: loadAI  r0, @z  => r2
h: mult     r1, r2 => r1
i: storeAI  r1 => r0,@w
```

S(n) =

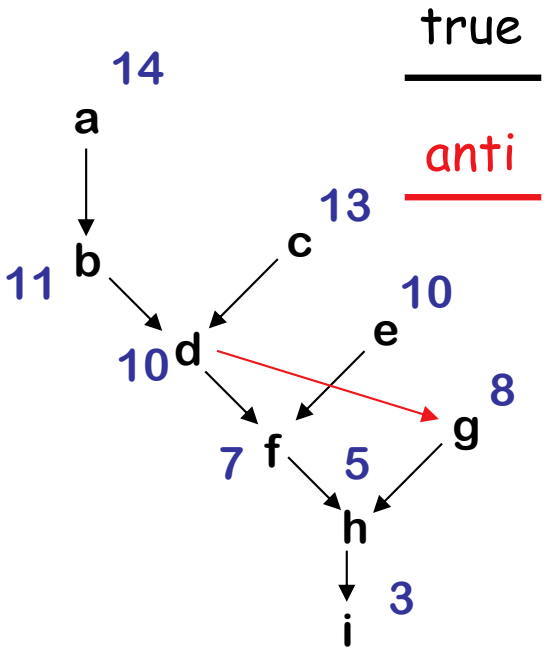
CYCLE = 0

READY - SET

ACTIVE - SET

The Generated Code

0 1 2 3 4 5 6 7 8 9 10 11 12



The Dependence Graph

(longest latency-weighted)

Next topic

Finishing instruction scheduling

Register Allocation EaC 13.1 – 13.3
(Top-down and Bottom-Up Allocation)