Lecture 18
List Scheduling

Reading: Chapter 10.3
Review: The Ideal Scheduling Outcome

- What prevents us from achieving this ideal?

Before

After

Time

N cycles

1 cycle
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
**Scheduling Roadmap**

**List Scheduling:**
- *within* a basic block

**Global Scheduling:**
- *across* basic blocks

**Software Pipelining:**
- *across* loop iterations
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

\[ x = a + b \]
\[ ... \]
\[ y = c + d \]
List Scheduling Algorithm: Inputs and Outputs

Algorithm reproduced from:


**Inputs:**
- Data Precedence Graph (DPG)

**Outputs:**
- Scheduled Code
- Cycle

<table>
<thead>
<tr>
<th>Scheduled Code</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0  I2</td>
<td>0</td>
</tr>
<tr>
<td>--- I1 I4</td>
<td>1</td>
</tr>
<tr>
<td>I3  I8 I6</td>
<td>2</td>
</tr>
<tr>
<td>I10 --- I11</td>
<td>3</td>
</tr>
<tr>
<td>I7  I9 I5</td>
<td>4</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Cycle</th>
<th>I2</th>
<th>I0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cycle 0 is highlighted.
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, ...

Carnegie Mellon
Representing Data Dependences: The Data Precedence Graph (DPG)

- Two different kinds of edges:
  - True “edges”: \( E \) (read-after-write)
  - “Anti-edges”: \( E' \) (write-after-read)

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

\[
priority(x) = \max(\forall l \in \text{leaves}(DPG) \forall p \in \text{paths}(x, \ldots, l) \sum_{p_i=x}^{l} \text{latency}(p_i))
\]
Computing Priorities (Cont.)

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

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

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

• 2 identical fully-pipelined FUs
• adds take 2 cycles; all other insts take 1 cycle
What if We Break Ties Differently?

\[
\begin{align*}
I0: & \quad a = 1 \\
I1: & \quad f = a + x \\
I2: & \quad b = 7 \\
I3: & \quad c = 9 \\
I4: & \quad g = f + b \\
I5: & \quad d = 13 \\
I6: & \quad e = 19; \\
I7: & \quad h = f + c \\
I8: & \quad j = d + y \\
I9: & \quad z = -1 \\
I10: & \quad \text{JMP L1}
\end{align*}
\]

- 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**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td></td>
<td>I2</td>
<td></td>
<td></td>
<td>I5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I1</td>
<td>I5</td>
<td>I1</td>
<td>I8</td>
<td>I7</td>
<td>I4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td>I8</td>
<td>I3</td>
<td>I7</td>
<td>I4</td>
<td>I1</td>
<td>I6</td>
<td>I8</td>
</tr>
<tr>
<td>I3</td>
<td></td>
<td>I5</td>
<td>I8</td>
<td>I7</td>
<td>I4</td>
<td>I2</td>
<td>I3</td>
</tr>
<tr>
<td>I4</td>
<td></td>
<td>I5</td>
<td>I8</td>
<td>I7</td>
<td>I4</td>
<td>I1</td>
<td>I6</td>
</tr>
<tr>
<td>I5</td>
<td></td>
<td>I5</td>
<td>I8</td>
<td>I7</td>
<td>I4</td>
<td>I2</td>
<td>I3</td>
</tr>
<tr>
<td>I6</td>
<td></td>
<td>I5</td>
<td>I8</td>
<td>I7</td>
<td>I4</td>
<td>I1</td>
<td>I6</td>
</tr>
<tr>
<td>I7</td>
<td></td>
<td>I5</td>
<td>I8</td>
<td>I7</td>
<td>I4</td>
<td>I2</td>
<td>I3</td>
</tr>
<tr>
<td>I8</td>
<td></td>
<td>I5</td>
<td>I8</td>
<td>I7</td>
<td>I4</td>
<td>I1</td>
<td>I6</td>
</tr>
<tr>
<td>I9</td>
<td></td>
<td>I5</td>
<td>I8</td>
<td>I7</td>
<td>I4</td>
<td>I1</td>
<td>I6</td>
</tr>
<tr>
<td>I10</td>
<td></td>
<td>I5</td>
<td>I8</td>
<td>I7</td>
<td>I4</td>
<td>I1</td>
<td>I6</td>
</tr>
</tbody>
</table>

- 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.
**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

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

<table>
<thead>
<tr>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDIa</td>
<td>LSL</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>LDIb</td>
<td>LDIC</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>LDIc</td>
<td>ADDa</td>
<td>----</td>
<td>2</td>
</tr>
<tr>
<td>ADDb</td>
<td>ADDc</td>
<td>----</td>
<td>3</td>
</tr>
<tr>
<td>ADDd</td>
<td>ADDI</td>
<td>STa</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>----</td>
<td>STb</td>
<td>5</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>STc</td>
<td>6</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>STd</td>
<td>7</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>STe</td>
<td>8</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>9</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>10</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>11</td>
</tr>
<tr>
<td>BR</td>
<td>----</td>
<td>----</td>
<td>12</td>
</tr>
</tbody>
</table>
Now Let’s Try Scheduling **Backward**

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

<table>
<thead>
<tr>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDIa</td>
<td>-----</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>ADDI</td>
<td>LSL</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>ADDd</td>
<td>LDId</td>
<td>----</td>
<td>2</td>
</tr>
<tr>
<td>ADDc</td>
<td>LDId</td>
<td>STe</td>
<td>3</td>
</tr>
<tr>
<td>ADDb</td>
<td>LDIa</td>
<td>STd</td>
<td>4</td>
</tr>
<tr>
<td>ADDa</td>
<td>-----</td>
<td>STc</td>
<td>5</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>STb</td>
<td>6</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>STa</td>
<td>7</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>CMP</td>
<td>-----</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>BR</td>
<td>-----</td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>
### Contrasting Forward vs. Backward List Scheduling

#### Forward

<table>
<thead>
<tr>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDIa</td>
<td>LSL</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>LDIb</td>
<td>LDIC</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>LDIc</td>
<td>ADDa</td>
<td>----</td>
<td>2</td>
</tr>
<tr>
<td>ADDb</td>
<td>ADDc</td>
<td>----</td>
<td>3</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDI</td>
<td>STa</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>----</td>
<td>STb</td>
<td>5</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>STc</td>
<td>6</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>STd</td>
<td>7</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>STE</td>
<td>8</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>9</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>10</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>11</td>
</tr>
<tr>
<td>BR</td>
<td>----</td>
<td>----</td>
<td>12</td>
</tr>
</tbody>
</table>

#### Backward

<table>
<thead>
<tr>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDIa</td>
<td>----</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>ADDI</td>
<td>LSL</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>ADDc</td>
<td>LDIC</td>
<td>----</td>
<td>2</td>
</tr>
<tr>
<td>ADDd</td>
<td>LDID</td>
<td>STe</td>
<td>3</td>
</tr>
<tr>
<td>ADDb</td>
<td>LDIA</td>
<td>STd</td>
<td>4</td>
</tr>
<tr>
<td>ADDa</td>
<td>----</td>
<td>STc</td>
<td>5</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>STb</td>
<td>6</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>STa</td>
<td>7</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>8</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>9</td>
</tr>
<tr>
<td>CMP</td>
<td>----</td>
<td>----</td>
<td>10</td>
</tr>
<tr>
<td>BR</td>
<td>----</td>
<td>----</td>
<td>11</td>
</tr>
</tbody>
</table>

- 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