Lecture 18
List Scheduling

Reading: Chapter 10.3

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: 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

List Scheduling Algorithm: Inputs and Outputs

<table>
<thead>
<tr>
<th>Inputs: Data Precedence Graph (DPG)</th>
<th>Machine Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of FUs:</td>
</tr>
<tr>
<td>I0</td>
<td>2 INT, 1 FP</td>
</tr>
<tr>
<td>I2</td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td></td>
</tr>
<tr>
<td>I6</td>
<td></td>
</tr>
<tr>
<td>I10</td>
<td></td>
</tr>
<tr>
<td>I11</td>
<td></td>
</tr>
<tr>
<td>I15</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output: 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: Algorithm reproduced from:

Example:

- I2 I0

Cycle

- 0
- 1
- 2
What Makes Life Interesting: Choice

Easy case:
- all ready instructions can be scheduled this cycle

![Diagram of instructions I5, I1, I7]

Interesting case:
- we need to pick a subset of the ready instructions

![Diagram of instructions I5, I1, I0, I2, I7 with question mark]

- 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?

![Data Dependence Graph (DPG) with nodes 10, 12, 11, 14, 16, 13, 18, 19]

# 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:
  - Code
    - true "edges": E
      - (read-after-write)
      - (I0, I1)
    - anti-edges": E'
      - (write-after-read)
      - (I1, I2)
  - 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

$$\text{priority}(x) = \max_y (\text{leaves}(\text{DPG}) \sum_{p \in \text{paths}(x, y)} \text{latency}(p))$$
Computing Priorities (Cont.)

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

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

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 decreasing 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

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

What if We Break Ties Differently?

- 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

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>I0</td>
</tr>
<tr>
<td>I2</td>
<td>I2</td>
</tr>
<tr>
<td>I1</td>
<td>I1</td>
</tr>
<tr>
<td>I3</td>
<td>I5</td>
</tr>
<tr>
<td>I9</td>
<td>I6</td>
</tr>
<tr>
<td>I7</td>
<td>I4</td>
</tr>
<tr>
<td>I8</td>
<td>I1</td>
</tr>
<tr>
<td>I10</td>
<td>I10</td>
</tr>
<tr>
<td>I10</td>
<td>I10</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT INT MEM</td>
<td>Cycle</td>
</tr>
<tr>
<td>LDIa LSL ----</td>
<td>1</td>
</tr>
<tr>
<td>ADDd ADDd ----</td>
<td>2</td>
</tr>
<tr>
<td>ADDd ADDd ----</td>
<td>3</td>
</tr>
<tr>
<td>ADDd ADDd STB</td>
<td>4</td>
</tr>
<tr>
<td>CMP ---- STB</td>
<td>5</td>
</tr>
<tr>
<td>---- ---- STB</td>
<td>6</td>
</tr>
<tr>
<td>---- ---- STB</td>
<td>7</td>
</tr>
<tr>
<td>---- ---- STB</td>
<td>8</td>
</tr>
<tr>
<td>---- ---- STB</td>
<td>9</td>
</tr>
<tr>
<td>---- ---- STB</td>
<td>10</td>
</tr>
<tr>
<td>---- ---- STB</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