List Scheduling

Todd C. Mowry
CS745: Optimizing Compilers

Review: The Ideal Scheduling Outcome

Before

After

Time

N cycles

1 cycle

What prevents us from achieving this ideal?

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

Trace 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

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

Algorithm reproduced from:

Inputs:
- Data Precedence Graph (DPG)
- Machine Parameters

Output:
- Scheduled Code
- Cycle

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?

### Example

- **Number 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:
  - true "edges": $E$
    - $e = (I_0, I_1)$ (read-after-write)
  - anti-edges: $E'$
    - $e' = (I_1, I_2)$ (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

\[
\text{priority}(x) = \max \{ |\text{depth}(x)\text{ in } DPG | \}_{x \in E} \sum_{e \in E} \text{latency}(p_i)
\]

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

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

cycle = 0;
ready-list = root nodes in DFG; 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

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
What if We Break Ties Differently?

10: \( a = 1 \)
11: \( f = a + x \)
12: \( b = 7 \)
13: \( c = 9 \)
14: \( g = f + b \)
15: \( d = 13 \)
16: \( e = 19 \)
17: \( h = f + c \)
18: \( j = d + y \)
19: \( z = -1 \)
20: \( \text{JMP L1} \)

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

Contrasting the Two Schedules

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
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

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>ADDa</td>
<td>LDIc</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>ADDb</td>
<td>LDId</td>
<td>STa</td>
<td>2</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDa</td>
<td>STb</td>
<td>3</td>
</tr>
<tr>
<td>ADDd</td>
<td>ADDb</td>
<td>STc</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>ADDc</td>
<td>STd</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>CMP</td>
<td>STe</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STf</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>BR</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>
</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>LSL</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>ADDa</td>
<td>LDIc</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>ADDb</td>
<td>LDId</td>
<td>STa</td>
<td>2</td>
</tr>
<tr>
<td>ADDc</td>
<td>ADDa</td>
<td>STb</td>
<td>3</td>
</tr>
<tr>
<td>ADDd</td>
<td>ADDb</td>
<td>STc</td>
<td>4</td>
</tr>
<tr>
<td>CMP</td>
<td>ADDc</td>
<td>STd</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>CMP</td>
<td>STe</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STf</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>BR</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>
</tbody>
</table>

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

Trace Scheduling:  • across basic blocks

Software Pipelining:  • across loop iterations

\[ x = a + b \]
\[ \ldots \]
\[ y = c + d \]

\[ x = a + b \]
\[ y = c + d \]