Lecture 18
List Scheduling & Global Scheduling

Reading: Chapter 10.3-10.4
Review: The Ideal Scheduling Outcome

- What prevents us from achieving this ideal?

**Before**

- Time
- N cycles

**After**

- 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
List Scheduling Algorithm: Inputs and Outputs

Algorithm reproduced from:


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

![Data Precedence Graph](image)

**Machine Parameters**
- # of FUs: 2 INT, 1 FP
- Latencies: add = 1 cycle, ...
- Pipelining: 1 add/cycle, ...

**Output:**

<table>
<thead>
<tr>
<th>Scheduled Code</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 12 ---</td>
<td>0</td>
</tr>
<tr>
<td>--- 11 14</td>
<td>1</td>
</tr>
<tr>
<td>13 18 16</td>
<td>2</td>
</tr>
<tr>
<td>110 --- 111</td>
<td>3</td>
</tr>
<tr>
<td>17 19 15</td>
<td>4</td>
</tr>
</tbody>
</table>

Todd C. Mowry
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
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, ...

I0 → I2 → I1 → I4
  I1 → I8 → I5
  I4 → I3 → I9
  $\lfloor$ 2 INT, 1 FP $\rfloor$
  $\lfloor$ add = 1 cycle, ...
  $\lfloor$ 1 add/cycle, ...

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

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

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

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

<table>
<thead>
<tr>
<th>Cycle</th>
<th>10</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>I0</td>
<td>I2</td>
</tr>
<tr>
<td>1</td>
<td>I1</td>
<td>I3</td>
</tr>
<tr>
<td>2</td>
<td>I5</td>
<td>I9</td>
</tr>
<tr>
<td>3</td>
<td>I4</td>
<td>I7</td>
</tr>
<tr>
<td>4</td>
<td>I8</td>
<td>I6</td>
</tr>
<tr>
<td>5</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td>I10</td>
<td></td>
</tr>
</tbody>
</table>
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

• 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: \( \text{JMP L1} \)

• 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

Forward Scheduling Priorities

Backward Scheduling Priorities
**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></th>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDla</td>
<td>LSL</td>
<td>-----</td>
<td>----</td>
<td>0</td>
</tr>
<tr>
<td>LDlb</td>
<td>LDlc</td>
<td>-----</td>
<td>----</td>
<td>1</td>
</tr>
<tr>
<td>LDld</td>
<td>ADDa</td>
<td>-----</td>
<td>----</td>
<td>2</td>
</tr>
<tr>
<td>ADDb</td>
<td>ADDc</td>
<td>-----</td>
<td>----</td>
<td>3</td>
</tr>
<tr>
<td>ADDd</td>
<td>ADDI</td>
<td>STa</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>-----</td>
<td>STb</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>STc</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>STd</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>STe</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>-----</td>
<td>-----</td>
<td>12</td>
<td></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
## Contrasting Forward vs. Backward List Scheduling

<table>
<thead>
<tr>
<th>Cycle</th>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INT</td>
<td>LSL</td>
<td>----</td>
<td>LDia</td>
</tr>
<tr>
<td>1</td>
<td>INT</td>
<td>LIdc</td>
<td>----</td>
<td>LDb</td>
</tr>
<tr>
<td>2</td>
<td>INT</td>
<td>ADDa</td>
<td>----</td>
<td>LDId</td>
</tr>
<tr>
<td>3</td>
<td>INT</td>
<td>ADDc</td>
<td>----</td>
<td>ADDb</td>
</tr>
<tr>
<td>4</td>
<td>INT</td>
<td>ADDD</td>
<td>STa</td>
<td>ADDd</td>
</tr>
<tr>
<td>5</td>
<td>INT</td>
<td>STb</td>
<td>----</td>
<td>CMP</td>
</tr>
<tr>
<td>6</td>
<td>INT</td>
<td>STc</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>INT</td>
<td>STd</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>INT</td>
<td>STe</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>INT</td>
<td>----</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>INT</td>
<td>----</td>
<td>----</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>INT</td>
<td>----</td>
<td>----</td>
<td>BR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>INT</th>
<th>INT</th>
<th>MEM</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INT</td>
<td>----</td>
<td>----</td>
<td>LDia</td>
</tr>
<tr>
<td>1</td>
<td>INT</td>
<td>LSL</td>
<td>----</td>
<td>ADDI</td>
</tr>
<tr>
<td>2</td>
<td>INT</td>
<td>LIdc</td>
<td>----</td>
<td>ADDD</td>
</tr>
<tr>
<td>3</td>
<td>INT</td>
<td>LIdb</td>
<td>STe</td>
<td>ADDc</td>
</tr>
<tr>
<td>4</td>
<td>INT</td>
<td>LDia</td>
<td>STd</td>
<td>ADDb</td>
</tr>
<tr>
<td>5</td>
<td>INT</td>
<td>ADDa</td>
<td>STc</td>
<td>ADDa</td>
</tr>
<tr>
<td>6</td>
<td>INT</td>
<td></td>
<td>STb</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>INT</td>
<td></td>
<td>STa</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>INT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>INT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>INT</td>
<td>CMP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>INT</td>
<td>BR</td>
<td></td>
<td></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 $\sim 2.5$
- 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
Assume each clock can execute 2 operations of any kind.

```
if (a==0) goto L

L:

c = b

e = d + d

B1:
LD R6 <- 0(R1)
nop
BEQZ R6, L

B2:
LD R7 <- 0(R2)
nop
ST 0(R3) <- R7

B3:
LD R8 <- 0(R4)
nop
ADD R8 <- R8,R8
ST 0(R5) <- R8
```
Result of Code Scheduling

LD R6 <- 0(R1) ; LD R8 <- 0(R4)
LD R7 <- 0(R2)
ADD R8 <- R8,R8 ; BEQZ R6, L

L: ST 0(R5) <- R8

B₁

ST 0(R5) <- R8 ; ST 0(R3) <- R7
B₃′

B₃
Terminology

Control equivalence:
- Two operations $o_1$ and $o_2$ are *control equivalent* if $o_1$ is executed if and only if $o_2$ is executed.

Control dependence:
- An op $o_2$ is *control dependent* on op $o_1$ if the execution of $o_2$ depends on the outcome of $o_1$.

Speculation:
- An operation $o$ is *speculatively* executed if it is executed before all the operations it depends on (control-wise) have been executed.
- Requirements:
  - does not raise an exception
  - satisfies data dependences
Code Motions

Goal: Shorten execution time probabilistically

Moving instructions up:
- Move instruction to a cut set (from entry)
- Speculation: even when not anticipated.

Moving instructions down:
- Move instruction to a cut set (from exit)
- May execute extra instruction
- Can duplicate code
A Note on Data Dependences

\[ a = 0 \quad \text{and} \quad a = 1 \]
General-Purpose Applications

- Lots of data dependences
- Key performance factor: memory latencies
- Move memory fetches up
  - Speculative memory fetches can be expensive
- Control-intensive: get execution profile
  - Static estimation
    - Innermost loops are frequently executed
      - back edges are likely to be taken
    - Edges that branch to exit and exception routines are not likely to be taken
  - Dynamic profiling
    - Instrument code and measure using representative data
A Basic Global Scheduling Algorithm

• Schedule innermost loops first
• Only upward code motion
• No creation of copies
• Only one level of speculation
Program Representation

• **A region in a control flow graph is:**
  – a set of basic blocks and all the edges connecting these blocks,
  – such that control from outside the region must enter through a single entry block.

• **A procedure is represented as a hierarchy of regions**
  – The whole control flow graph is a region
  – Each natural loop in the flow graph is a region
  – Natural loops are hierarchically nested

• **Schedule regions from inner to outer**
  – treat inner loop as a black box unit
    * can schedule around it but not into it
  – ignore all the loop back edges → get an acyclic graph
Algorithm

Compute data dependences;
For each region from inner to outer {
    For each basic block B in prioritized topological order {
        CandBlocks = ControlEquiv\{B\} U
            Dominated-Successors\{ControlEquiv\{B\}\};
        CandInsts = ready operations in CandBlocks;
        For (t = 0, 1, ... until all operations from B are scheduled) {
            For (n in CandInst in priority order) {
                if (n has no resource conflicts at time t) {
                    S(n) = < B, t >
                    Update resource commitments
                    Update data dependences
                }
            }
        }
    }
    Update CandInsts;
}}

Priority functions: non-speculative before speculative
Extensions

• Prepass before scheduling: loop unrolling

• Especially important to move operation up loop back edges

\[\text{Diagram showing loop unrolling and back edges.}\]
Summary

- **Global scheduling**
  - Legal code motions
  - Heuristics