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  After

Time

1 cycle

N cycles

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

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)
- Machine Parameters

<table>
<thead>
<tr>
<th># of FUs:</th>
<th>Latencies:</th>
<th>Pipelining:</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 INT, 1 FP</td>
<td>add = 1 cycle, ...</td>
<td>1 add/cycle, ...</td>
</tr>
</tbody>
</table>

Output:
- Scheduled Code
- Cycle

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

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

Intuition Behind Priorities

- Intuitively, what should the priority correspond to?
- What factors are used to compute it?
  - data dependences?
  - machine parameters?
Representing Data Dependences: The Data Precedence Graph (DPG)

- Two different kinds of edges:
  - **Code**
  - **true "edges":** $E$
    - (read-after-write)
    - $I_0: x = 1$
    - $I_1: y = x$
    - $I_2: x = 2$
    - $I_3: z = x$
  - **anti-edges**: $E'$
    - (write-after-read)
    - $I_e = (I_0, I_1)$
    - $I_e' = (I_1, I_2)$
    - $I_e = (I_2, I_3)$

- 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

$$\text{priority}(x) = \max_{(y \in \text{leaves}(DPG))} \sum_{p \in \text{paths}(x, y)} \text{latency}(p_i)$$

Computing Priorities (Cont.)

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

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

List Scheduling Algorithm

```plaintext
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**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>I0: a = 1</th>
<th>I1: f = a + x</th>
<th>I2: b = 7</th>
<th>I3: c = 9</th>
<th>I4: g = f + b</th>
<th>I5: d = 13</th>
<th>I6: e = 19</th>
<th>I7: h = f + c</th>
<th>I8: j = d + y</th>
<th>I9: z = -1</th>
<th>I10: JMP L1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>10</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

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

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

Introduction to Global Scheduling

Assume each clock can execute 2 operations of any kind.

if (a==0) goto L

L: c = b

... 

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

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

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

LD R8 <- 0(R4)
nop
ADD R8 <- R8,R8
ST 0(R5) <- R8

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

Result of Code Scheduling

... 

B1

B2

B3

B4

L: ST 0(R5) <- R8

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

L: ST 0(R5) <- R8 ; ST 0(R3) <- R7
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 operation $o_2$ is control dependent on operation $o_1$ 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

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) ∪ Dominated-Successors(ControlEquiv(B));
    - CandInsts = ready operations in CandBlocks;
    - For (t = 0, 1, ... until all operations from B are scheduled)
      - 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

Summary

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