Lecture 15
Instruction Scheduling

I. Hardware Support for Parallel Execution
II. Constraints on Scheduling
III. List Scheduling

[ALSU 10.1-10.3]

The Goal of Instruction Scheduling

• Assume that the remaining instructions are all essential
  (otherwise, earlier passes would have eliminated them)
• How can we perform this fixed amount of work in less time?
  Answer: execute the instructions in parallel

Time

\[
\begin{align*}
a &= 1 + x; \\
b &= 2 + y; \\
c &= 3 + z;
\end{align*}
\]

Optimization: What’s the Point? (A Quick Review)

Machine-Independent Optimizations:
  – e.g., constant propagation & folding, redundancy elimination, dead-code elimination, etc.
  – Goal: eliminate work

Machine-Dependent Optimizations:
  – register allocation
    • Goal: reduce cost of accessing data
  – instruction scheduling
    • Goal: ???
    • ...

I. Hardware Support for Parallel Execution

• Three forms of parallelism are found in modern machines:
  – Pipelining
  – Superscalar Processing
  – Multiprocessing

Automatic Parallelization
covered later in class
**Pipelining**

**Basic idea:**
- break instruction into stages that can be overlapped

**Example:** simple 5-stage pipeline from early RISC machines

1 instruction

<table>
<thead>
<tr>
<th>IF</th>
<th>RF</th>
<th>EX</th>
<th>ME</th>
<th>WB</th>
</tr>
</thead>
</table>

**Time**

- IF = Instruction Fetch
- RF = Decode & Register Fetch
- EX = Execute on ALU
- ME = Memory Access
- WB = Write Back to Register File

**Beyond 5-Stage Pipelines: Even More Parallelism**

- Should we simply make pipelines deeper and deeper?
  - registers between pipeline stages have fixed overheads
  - hence diminishing returns with more stages (Amdahl's Law)
  - value of pipe stage unclear if < time for integer add

  However, many consumers think "performance = clock rate"
  - perceived need for higher clock rates -> deeper pipelines
  - e.g., Pentium 4 processor had a 20-stage pipeline
Beyond Pipelining: “Superscalar” Processing

- Basic Idea:
  - multiple (independent) instructions can proceed simultaneously through the same pipeline stages
- Requires additional hardware
  - example: “Execute” stage

Superscalar Pipeline Illustration

Original (scalar) pipeline:
- Only one instruction in a given pipe stage at a given time

Superscalar pipeline:
- Multiple instructions in the same pipe stage at the same time

II. Constraints on Scheduling

1. Hardware Resources
2. Data Dependences
3. Control Dependences

Constraint #1: Hardware Resources

- Processors have finite resources, and there are often constraints on how these resources can be used.

Examples:
- Finite issue width
- Limited functional units (FUs) per given instruction type
- Limited pipelining within a given functional unit (FU)
**Finite Issue Width**

- Prior to superscalar processing:
  - Processors only "issued" one instruction per cycle
- Even with superscalar processing:
  - Limit on total # of instructions issued per cycle

### Finite Issue Width Diagram

<table>
<thead>
<tr>
<th>Time</th>
<th>Issue Width = infinite</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Issue Width = 4</th>
</tr>
</thead>
</table>

\[
\text{Time} \geq \frac{N}{4}
\]

**Limited FUs per Instruction Type**

- e.g., a 4-way superscalar might only be able to issue up to 2 integer, 1 memory, and 1 floating-point insts per cycle

### Limited FUs per Instruction Type Diagram

**Original Code**

- Integer
- Memory
- Floating-Point

**Unconstrained**

- Integer
- Memory
- Floating-Point

**More Realistic**

- Integer
- Memory
- Floating-Point

**Bottleneck**

- Empty Slot

**Constraints on Scheduling**

1. Hardware Resources
2. Data Dependences
3. Control Dependences

**Limited Pipelining within a Functional Unit**

- e.g., only 1 new floating-point division once every 2 cycles

### Limited Pipelining within a Functional Unit Diagram

**Original Code**

- Integer
- Memory
- Floating-Point

**Schedule with Limited Pipelining**

- Integer
- Memory
- Floating-Point

- Empty Slot
Constraint #2: Data Dependences

- If we read or write a data location "too early", the program may behave incorrectly.

(Assume that initially, x = 0.)

\[ x = 1; \]
\[ y = x; \]
\[ x = 1; \]
\[ y = x; \]
\[ x = 2; \]
\[ y = x; \]
\[ x = 1; \]
\[ y = x; \]

Read-after-Write (*"True" dependence)  
Write-after-Write (*"Output" dependence)  
Write-after-Read (*"Anti" dependence)

Fundamental (no simple fix)  
Can potentially fix through renaming.

Given Ambiguous Data Dependences, What To Do?

- Conservative approach: don’t reorder instructions
  - ensures correct execution
  - but may suffer poor performance
- Aggressive approach?
  - is there a way to safely reorder instructions?

Why Data Dependences are Challenging

\[ x = a[i]; \]
\[ *p = 1; \]
\[ y = *q; \]
\[ *r = z; \]

- which of these instructions can be reordered?
- ambiguous data dependences are very common in practice
  - difficult to resolve, despite fancy pointer analysis [next lecture]

Hardware Limitations: Multi-cycle Execution Latencies

- Simple instructions often "execute" in one cycle
  - (as observed by other instructions in the pipeline)
  - e.g., integer addition
- More complex instructions may require multiple cycles
  - e.g., integer division, square-root
  - cache misses!
- These latencies, when combined with data dependencies, can result in non-trivial critical path lengths through code
Constraints on Scheduling

1. Hardware Resources
2. Data Dependences
3. Control Dependences

Constraint #3: Control Dependences

- What do we do when we reach a conditional branch?
  - choose a “frequently-executed” path?
  - choose multiple paths?

Scheduling Constraints: Summary

- 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

III. List Scheduling

- The most common technique for scheduling instructions within a basic block

Basic block scheduling doesn’t need to worry about:
  - control flow [future lecture]

Does 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

Output:
- Scheduled Code
- Cycle

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**
  
  \[ \begin{align*}
  I_0 &: x = 1; \\
  I_1 &: y = x; \\
  I_2 &: x = 2; \\
  I_3 &: z = x; 
  \end{align*} \]

  **DPG**
  
  \[ \begin{align*}
  e &= (I_0, I_1) \\
  e' &= (I_2, I_3) 
  \end{align*} \]

- 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

\[
\text{priority}(x) = \max \left( \forall_{e \in E(DPG)} \forall_{p \in \text{paths}(x, \ldots)} \sum_{p_i=x}^{\text{latency}(p_i)} \right)
\]

List Scheduling Algorithm

\[
cycle = 0; \\
\text{ready-list} = \text{root nodes in DPG}; \\
\text{inflight-list} = \emptyset.
\]

while (\(|\text{ready-list}|+|\text{inflight-list}| > 0\) && an issue slot is available) {
  for \( op = \text{all nodes in ready-list in descending priority order} \) {
    if (an FU exists for \( op \) to start at \( cycle \)) {
      remove \( op \) from \( \text{ready-list} \) and add to \( \text{inflight-list} \);
      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 \( \text{ready-list} \);}
  }
  \( cycle = cycle + 1; \)
  for \( op = \text{all nodes in \( \text{inflight-list} \)}\)
    if (\( op \) finishes at time \( cycle \) ) {
      remove \( op \) from \( \text{inflight-list} \);
      check nodes waiting for \( op \) & add to \( \text{ready-list} \) if all operands available;}
}
}
### List Scheduling 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

### 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
**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>LDI</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>LDIb</td>
<td>LDIc</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>ADD</td>
<td>LSL</td>
<td>...</td>
<td>2</td>
</tr>
<tr>
<td>ADD</td>
<td>LSL</td>
<td>...</td>
<td>3</td>
</tr>
<tr>
<td>ADD</td>
<td>LSL</td>
<td>...</td>
<td>4</td>
</tr>
<tr>
<td>ADD</td>
<td>LSL</td>
<td>...</td>
<td>5</td>
</tr>
<tr>
<td>ADD</td>
<td>LSL</td>
<td>...</td>
<td>6</td>
</tr>
<tr>
<td>ADD</td>
<td>LSL</td>
<td>...</td>
<td>7</td>
</tr>
<tr>
<td>ADD</td>
<td>LSL</td>
<td>...</td>
<td>8</td>
</tr>
<tr>
<td>CMP</td>
<td>STa</td>
<td>STb</td>
<td>9</td>
</tr>
<tr>
<td>CMP</td>
<td>STa</td>
<td>STb</td>
<td>10</td>
</tr>
<tr>
<td>CMP</td>
<td>STa</td>
<td>STb</td>
<td>11</td>
</tr>
<tr>
<td>CMP</td>
<td>STa</td>
<td>STb</td>
<td>12</td>
</tr>
<tr>
<td>BR</td>
<td>STa</td>
<td>STb</td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>STa</td>
<td>STb</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

<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>LSL</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>2</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>3</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>4</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>5</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>6</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>7</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>8</td>
</tr>
<tr>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
<td>9</td>
</tr>
<tr>
<td>CMP</td>
<td>STa</td>
<td>STb</td>
<td>10</td>
</tr>
<tr>
<td>CMP</td>
<td>STa</td>
<td>STb</td>
<td>11</td>
</tr>
<tr>
<td>CMP</td>
<td>STa</td>
<td>STb</td>
<td>12</td>
</tr>
<tr>
<td>BR</td>
<td>STa</td>
<td>STb</td>
<td></td>
</tr>
<tr>
<td>BR</td>
<td>STa</td>
<td>STb</td>
<td></td>
</tr>
</tbody>
</table>

### Contrasting Forward vs. Backward List Scheduling
- 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

Efficient Instruction Scheduling for a Pipelined Architecture

Phillip B. Gibbons* & Steven S. Muchnick**

 Hewlett-Packard Laboratories
 1901 Page Mill Road
 Palo Alto, CA 94304-1181

Abstract

As part of an effort to develop an optimizing compiler for a pipelined architecture, a code reorganization algorithm has been developed that significantly reduces the number of necessary pipeline interlocks. In a pass after code generation, the algorithm uses a dag representation to hierarchically schedule the instructions in each basic block.

Previous algorithms for reducing pipeline interlocks have had worst-case runtimes of at least $O(n^3)$. By using a dag representation which prevents scheduling deadlocks and a selection method that requires no lookahead, the resulting algorithm reorganizes instructions almost as effectively in practice, while having an $O(n \log n)$ worst-case runtime.

1. Introduction

The architecture we have studied has many features which

Looking Ahead

- Monday: Pointer Analysis
  [ALSU 12.4, 12.6-12.7]
- Wednesday: Dynamic Code Optimization
- Friday: No class
- Following Monday & Wednesday: "Recent Research on Optimization"
  - Student-led discussions, in groups of 2, with 20 minutes/group
  - Read 3 papers on a topic, and lead a discussion in class