Lecture 9

Instruction Scheduling

I. Basic Block Scheduling
II. Global Scheduling (for Non-Numeric Code)

Reading: Chapter 10.3 - 10.4
Who Schedules

- Compiler
- Assembler
- Hardware

\{, \}

Call

Movi all, 7
Scheduling Constraints

• **Data dependences**
  – The operations must generate the *same results* as the corresponding ones in the original program.

• **Control dependences**
  – All the operations executed in the original program must be executed in the optimized program.

• **Resource constraints**
  – No over-subscription of resources.
Data Dependence

• Must maintain order of accesses to potentially same locations
  – True dependence: write -> read (RAW hazard)
    \[ a = \ldots \]
    \[ = a \]
  – Output dependence: write -> write (WAW hazard)
    \[ a = \ldots \]
    \[ a = \ldots \]
  – Anti-dependence: read -> write (WAR hazard)
    \[ = a \]
    \[ a = \ldots \]

• Data Dependence Graph
  – Nodes: operations
  – Edges: \[ n_1 \rightarrow n_2 \] if \( n_2 \) is data dependent on \( n_1 \)
    – labeled by the execution length of \( n_1 \)
Analysis on Memory Variables

• **Undecidable in general**
  
  ```plaintext
  read x; read y;
  A[x] = ...
  ... = A[y]
  ```

• Two memory accesses can potentially be the same unless proven otherwise

• **Classes of analysis:**
  - **simple:** base+offset1 = base+offset2 ?
  - "**data dependence analysis**":
    - Array accesses whose indices are affine expressions of loop indices
      ```plaintext
      ```
  - **interprocedural analysis:** global = parameter?
  - **pointer analysis:** pointer1 = pointer2?
  - **language rules:**
    - int *a; float *b; *a=...; *b=...
    - int *restrict p;

• **Data dependence analysis is useful for many other purposes**
Aside

- Can these be reordered

for i

\[
\begin{align*}
A[i] &= A[i] + 1; \\
A[i+1] &= A[i+1] + 1;
\end{align*}
\]

LD R2 <- 0(R1)
ADDI R2 <- R2, 1
ST (R1) <- R2
ADDI R1 <- R1, 4

LD R3 <- 0(R1)
ADDI R3 <- R3, 1
ST (R1) <- R3
ADDI R1 <- R1, 4

LD R3 <- 4(R1)
Aside 2

- Can these be reordered

```plaintext
for i
    sum = sum + a[i];
    sum = sum + a[i+1];
LD R2 -< 0(R1++)
ADD R3 -< R3, R2
LD R2 -< 0(R1++)
ADD R3 -< R3, R2
LD R2 -< 0(R1++)
LD R4 -< 0(R1++)
ADD R3 -< R3, R2
ADD R3 -< R3, R4
```

sum2 = 0.0
for i
    sum = sum + a[i];
    sum2 = sum2 + a[i+1];
sum = sum + sum2
Resource Constraints

- Each instruction type has a resource reservation table

  Functional units

<table>
<thead>
<tr>
<th>Time</th>
<th>ld</th>
<th>st</th>
<th>alu</th>
<th>fmpy</th>
<th>fadd</th>
<th>br</th>
<th>...</th>
<th>div</th>
<th>alu2</th>
<th>il</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Pipelined functional units: occupy only one slot
- Non-pipelined functional units: multiple time slots
- Instructions may use more than one resource
- Multiple units of same resource
- Limited instruction issue slots
  - may also be managed like a resource
Example of a Machine Model

• Each machine cycle can execute 2 operations

• 1 ALU operation or branch operation
  \[ \text{Op dst,src1,src2} \] executes in 1 clock

• 1 load or store operation
  \[ \text{LD dst, addr} \] result is available in 2 clocks pipelined: can issue LD next clock
  \[ \text{ST src, addr} \] executes in 1 clock cycle
Basic Block Scheduling

LD R2 \(\leftarrow\) 0(R1)

ST 4(R1) \(\leftarrow\) R2

LD R3 \(\leftarrow\) 8(R1)

ADD R3 \(\leftarrow\) R3,R4

ADD R3 \(\leftarrow\) R3,R2

ST 12(R1) \(\leftarrow\) R3

ST 0(R7) \(\leftarrow\) R7
With Resource Constraints

• NP-complete in general → Heuristics time!

• List Scheduling:

  \[ \text{READY} = \text{nodes with 0 predecessors} \]

  Loop until \text{READY} is empty {

  Let \( n \) be the node in \text{READY} with \textbf{highest priority}

  Schedule \( n \) in the earliest slot
  that \textit{satisfies precedence + resource constraints}

  Update predecessor count of \( n \)'s successor nodes
  Update \text{READY}

  
  }

\[\text{M. Lam}\]
Basic Block Scheduling

LD R2 <- 0(R1)

ST 4(R1) <- R2

LD R3 <- 8(R1)

ADD R3 <- R3, R4

ADD R3 <- R3, R2

ST 12(R1) <- R3

ST 0(R7) <- R7

Ready: i1, i3

<table>
<thead>
<tr>
<th>T</th>
<th>ALU</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>i3</td>
</tr>
<tr>
<td>1</td>
<td>i1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>i4</td>
</tr>
<tr>
<td>3</td>
<td>i5</td>
<td>i2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>i6</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>i7</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
List Scheduling

• **Scope**: **DAGs**
  - Schedules operations in *topological* order
  - Never backtracks

• **Variations**:
  - *Priority function* for node *n*
    - *critical path*: max clocks from *n* to any node
    - resource requirements
    - source order
List Scheduling: Backwards

List Scheduling:

\[ \text{READY} = \text{nodes with 0 followers} \]

Loop until READY is empty {

Let \( n \) be the node in READY with \text{highest priority} \\

Schedule \( n \) in the latest slot \\
that satisfies precedence + resource constraints \\

Update follower count of \( n \)'s predecessor nodes \\
Update READY

}
Basic Block Scheduling

LD R2 ← 0(R1)

ST 4(R1) ← R2

LD R3 ← 8(R1)

ADD R3 ← R3,R4

ADD R3 ← R3,R2

ST 12(R1) ← R3

ST 0(R7) ← R7

Ready:
i7

<table>
<thead>
<tr>
<th>T</th>
<th>ALU</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>i3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i5</td>
<td>i2</td>
</tr>
<tr>
<td>5</td>
<td>i6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>i7</td>
<td></td>
</tr>
</tbody>
</table>
## Forward Versus Backwards

- Which is better?

<table>
<thead>
<tr>
<th>T</th>
<th>ALU</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>i3</td>
</tr>
<tr>
<td>1</td>
<td>i1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i5</td>
<td>i2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>i6</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>i7</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th>ALU</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>i3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i5</td>
<td>i2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>i6</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>i7</td>
</tr>
</tbody>
</table>
II. Introduction to Global Scheduling

Assume each clock can execute 2 operations of any kind.

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

L:
    e = d + d

if (p){
    load *p
}
```

Diagram:

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

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

- `B3`: 
  - `LD R8 <- 0(R4)`
  - `stall`
  - `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

• Note duplicated store
**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.
- Requirement: Raises no 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 Updating Data Dependences

\[
\begin{align*}
&\text{a} = 1 \\
&\text{a} = 0 \\
&\text{a} = 1 \\
&\ldots = \text{a}
\end{align*}
\]
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
**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
*Program Representation*

- **A region in a control flow graph is either:**
  - a reducible loop,
  - the entire function

- **A function 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 and can't schedule any CandInst in t) {
            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;
        }
    }
    Update CandInsts;
}}

Priority functions: non-speculative before speculative
Alternative: Hyperblock Scheduling

- Hyperblock: A set of basic blocks with a single entry
Hyperblock Scheduling

- Use a heuristic to select frequently executed blocks
- Tail duplication to ensure a single entry
- Node Splitting (optional but has advantages)
- If conversion
- Promotion (speculation)
- Instruction Merging (PRE)
- Scheduling

```
br
ld a7, 0(a8)   cmp b
ld b, a7, 0(a8)
ld_noexc a7, 0(a8)
cmp b
conditionally commit ld
```
Basic Algorithm Versus Hyperblock

- Basic algorithm designed for a machine with very limited parallelism
  - Hyperblock designed for a machine that has lots of parallelism
- Hyperblock assumes predicated instruction set
- Neither designed for numerical/media code nor for dynamically scheduled machines