Instruction Scheduling

List Scheduling, Trace Scheduling, Loop Unrolling & Software Pipelining

Copyright 2015, Pedro C. Diniz, all rights reserved.
Students enrolled in the Compilers class at the University of Southern California (USC) have explicit permission to make copies of these materials for their personal use.
Outline

• Overview of Instruction Scheduling
• List Scheduling
• Resource Constraints
• Interaction with Register Allocation
• Scheduling across Basic Blocks
• Trace Scheduling
• Scheduling for Loops
• Loop Unrolling
• Software Pipelining
Simple Execution Model

- 5 Stage pipe-line

<table>
<thead>
<tr>
<th>fetch</th>
<th>decode</th>
<th>execute</th>
<th>memory</th>
<th>writeback</th>
</tr>
</thead>
</table>

- Fetch: get the next instruction
- Decode: figure-out what that instruction is
- Execute: Perform ALU operation
  - address calculation in a memory op
- Write Back: write the results back
Simple Execution Model

.time

Inst 1

<table>
<thead>
<tr>
<th>IF</th>
<th>DE</th>
<th>EXE</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
</table>

Inst 2

<table>
<thead>
<tr>
<th>IF</th>
<th>DE</th>
<th>EXE</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
</table>
Simple Execution Model

- IF
- DE
- EXE
- MEM
- WB

Inst 1

Inst 2

Inst 1

Inst 2

Inst 3

Inst 4
Handling Branch and Jump Instructions

- Does not know the location of the next instruction until later
  - after DE in jump instructions
  - after EXE in branch instructions
Handling Branch and Jump Instructions

• Does not know the location of the next instruction until later
  – after DE in jump instructions
  – after EXE in branch instructions

• What to do with the middle 2 instructions?

<table>
<thead>
<tr>
<th>Branch</th>
<th>IF</th>
<th>DE</th>
<th>EXE</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
<td>IF</td>
<td>DE</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>???</td>
<td>IF</td>
<td>DE</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst</td>
<td>IF</td>
<td>DE</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Handling Branch and Jump Instructions

• What to do with the middle 2 Instructions?
• Delay the Action of the Branch (*Delay Slots*)
  – Make branch affect only after two instructions
  – Two instructions after the branch get executed regardless of the branch
Constraints On Scheduling

• Data Dependences
  – Inherent in the code

• Control Dependences
  – Inherent in the code

• Resource Constraints

• Sometimes we can Mitigate these Issues
  – Code restructuring
  – Instruction Scheduling
Data Dependency between Instructions

• If two instructions access the same variable (i.e. the same memory location), they can be dependent

• Kind of Dependences
  – True: \( \text{write} \rightarrow \text{read} \)
  – Anti: \( \text{read} \rightarrow \text{write} \)
  – Output: \( \text{write} \rightarrow \text{write} \)
  – Input: \( \text{read} \rightarrow \text{read} \)

• What to do if two Instructions are Dependent?
  – The order of execution cannot be reversed
  – Reduce the possibilities for scheduling
Representing Dependences

- Using a dependence DAG, one per Basic Block
- Nodes are instructions, edges represent dependences

1: \( r_2 = *(r_1 + 4) \)
2: \( r_3 = *(r_1 + 8) \)
3: \( r_4 = r_2 + r_3 \)
4: \( r_5 = r_2 - 1 \)
Representing Dependences

• Using a dependence DAG, one per Basic Block
• Nodes are instructions, edges represent dependences

1: \( r2 = *(r1 + 4) \)
2: \( r3 = *(r1 + 8) \)
3: \( r4 = r2 + r3 \)
4: \( r5 = r2 - 1 \)

• Edge is labeled with Latency:
  \[ v(i \rightarrow j) = \text{delay required between initiation times of } i \text{ and } j \text{ minus the execution time required by } i \]
Resource Constraints

- Modern Machines Have Many Resource Constraints
- Superscalar Architectures:
  - Can Execute few Operations Concurrently
  - But have constraints
  - Example:
    - 1 integer operation
      \[ \text{ALUop dest, src1, src2} \]  # in 1 clock cycle
      In parallel with
    - 1 memory operation
      \[ \text{LD dst, addr} \]  # in 2 clock cycles
      \[ \text{ST src, addr} \]  # in 1 clock cycle
Outline

• Overview of Instruction Scheduling
• List Scheduling
• Resource Constraints
• Interaction with Register Allocation
• Scheduling across Basic Blocks
• Trace Scheduling
• Scheduling for Loops
• Loop Unrolling
• Software Pipelining
List Scheduling Algorithm

• Idea:
  – Do a Topological Sorting of the Dependence DAG
  – Consider when an instruction can be scheduled without causing a stall
  – Schedule the instruction if it causes no stall and all its predecessors are already scheduled

• Optimal List Scheduling is NP-complete
  – Use Heuristics when necessary
List Scheduling Algorithm

- Create a dependence DAG of a Basic Block
- Topological Sorting
  
  READY List = nodes with no predecessors
  Loop until READY list is Empty
    Schedule each node in READY list when no stalling
    Update READY list
  end Loop
Heuristics for Selection

- Heuristics for selecting from the READY List
  - pick a node with the longest path to a leaf in the dependence graph
  - pick a node with most immediate successors
  - pick a node that can go to a less busy pipeline (in a superscalar)
Heuristics for Selection

• Pick a node with the longest path to a leaf in the dependence graph

• Algorithm (for node x)
  – If no successors then $d_x = 0$
  – else $d_x = \text{MAX}(d_y + c_{xy})$ for all successors y of x
  – reverse breadth-first visitation order
Heuristics for Selection

• Pick a node with most immediate successors
  – Rationale: Highest-degree will mean “solve” the most dependences

• Algorithm (for node x):
  – $f_x = \text{number of successors of } x$
Example

1 → 2
1 → 7
7 → 8 → d=0
2 → 3 → 4 → 5 → d=0
7 → 6 → 4
3
4
5
d=0

21
Example

1 \rightarrow 2 \rightarrow 3 \rightarrow 4
2 \rightarrow 7 \rightarrow 8 \rightarrow 9
3 \rightarrow 7 \rightarrow 8 \rightarrow 9
4 \rightarrow 7 \rightarrow 8 \rightarrow 9
5 \rightarrow 7 \rightarrow 8 \rightarrow 9
6 \rightarrow 7 \rightarrow 8 \rightarrow 9
7 \rightarrow 8 \rightarrow 9

\begin{align*}
\text{d} &= 0 \\
\text{d} &= 0 \\
\text{d} &= 0 \\
\text{d} &= 3 \\
\text{d} &= 0 \\
\text{d} &= 0 \\
\text{d} &= 3 \\
\text{d} &= 3
\end{align*}
Example

1 → 2: d=4
2 → 7: d=3
7 → 8: d=0
7 → 9: d=0
3 → 4: d=3
6 → 5: d=0

Example

1 \(d=5\) \(f=1\)

2 \(d=4\) \(f=1\)

3 \(d=0\) \(f=0\)

4 \(d=3\) \(f=1\)

5 \(d=0\) \(f=0\)

6 \(d=7\) \(f=1\)

7 \(d=3\) \(f=2\)

8 \(d=0\) \(f=0\)

9 \(d=0\) \(f=0\)
READY = \{ \}
Example

\[
\text{READY} = \{ 1, 3, 4, 6 \} 
\]

\[
\begin{align*}
1 & \quad d=5 \quad f=1 \\
2 & \quad d=4 \quad f=1 \\
3 & \quad d=0 \quad f=0 \\
4 & \quad d=3 \quad f=1 \\
5 & \quad d=0 \quad f=0 \\
6 & \quad d=7 \quad f=1 \\
7 & \quad d=3 \quad f=2 \\
8 & \quad d=0 \quad f=0 \\
9 & \quad d=0 \quad f=0 
\end{align*}
\]
Example

READY = \{ 6, 1, 4, 3 \}
READY = \{ 6, 1, 4, 3 \}
Example

READY = \{ 1, 4, 3 \}
Example

READY = \{ 1, 4, 3 \}
Example

READY = \{ 1, 4, 3 \}
Example

\[
\text{READY} = \{ 4, 3 \} \]

\[
\begin{array}{c}
\text{Node} & d & f \\
1 & 5 & 1 \\
2 & 4 & 1 \\
3 & 0 & 0 \\
4 & 3 & 1 \\
5 & 0 & 0 \\
6 & 7 & 1 \\
7 & 3 & 2 \\
8 & 0 & 0 \\
9 & 0 & 0 \\
\end{array}
\]
READY = { 2, 4, 3 }
Example

\[ \text{READY} = \{2, 4, 3\} \]
READY = \{ 2, 4, 3 \}
Example

\[ \text{READY} = \{4, 3\} \]
Example

\[ \text{READY} = \{ 4, 3 \} \]
READY = \{ 7, 4, 3 \}
READY = \{ 7, 4, 3 \}
Example

\[
\text{READY} = \{7, 4, 3\}
\]
Example

READY = { 7, 4, 3 }
READY = { 7, 4, 3 }
READY = \{ 7, 3 \}
Example

READY = { 7, 3, 5 }
Example

```plaintext
READY = \{ 7, 3, 5 \}
```
READY = \{ 3, 5 \}
Example

\[\text{READY} = \{3, 5\}\]
Example

READY = \{3, 5, 8, 9\}
Example

READY = \{3, 5, 8, 9\}
Example

\[
\text{READY} = \{3, 5, 8, 9\}
\]
READY = \{ 5, 8, 9 \}
Example

READY = { 5, 8, 9 }

6 1 2 4 7 3
Example

\[
\begin{align*}
\text{READY} &= \{ 5, 8, 9 \} \\
1 &\quad \text{d=5} \quad \text{f=1} \\
2 &\quad \text{d=4} \quad \text{f=1} \\
3 &\quad \text{d=0} \quad \text{f=0} \\
4 &\quad \text{d=3} \quad \text{f=1} \\
5 &\quad \text{d=0} \quad \text{f=0} \\
6 &\quad \text{d=7} \quad \text{f=1} \\
7 &\quad \text{d=3} \quad \text{f=2} \\
8 &\quad \text{d=0} \quad \text{f=0} \\
9 &\quad \text{d=0} \quad \text{f=0}
\end{align*}
\]
Example

READY = \{ 5, 8, 9 \}
Example

\[
\text{READY} = \{ 8, 9 \}
\]
Example

READY = \{ 8, 9 \}

\[
\begin{array}{c}
1 \quad d=5 \\
2 \quad d=4 \\
3 \quad d=0 \\
4 \quad d=3 \\
5 \\
6 \quad d=7 \\
7 \quad d=3 \\
8 \quad d=0 \\
9 \quad d=0 \\
\end{array}
\]

f: 1 0 1 1 0 0 2 1
Example

READY = \{ 8, 9 \}
Example

\[
\text{READY} = \{8, 9\}
\]
Example

\[ \text{READY} = \{8, 9\} \]
READY = { 9 }
Example

\[ \text{READY} = \{ 9 \} \]
Example

READY = \{ 9 \}
Example

READY = { 9 }

6 1 2 4 7 3 5 8 9
Example

READY = {
}

6 1 2 4 7 3 5 8 9
Example

READY = { }
Outline

• Overview of Instruction Scheduling
• List Scheduling
• Resource Constraints
• Interaction with Register Allocation
• Scheduling across basic blocks
• Trace Scheduling
• Scheduling for Loops
• Loop Unrolling
• Software Pipelining
List Scheduling with Resource Constraints

• Create a Dependence DAG of a Basic Block

• Topological Sort
  READY = nodes with no predecessors
  Loop until READY list is empty
    Let n ∈ READY list be the node with the highest priority
    Schedule n in the earliest slot that satisfies precedence + resource constraints
  Update READY list
Constraints of a Superscalar Processor

- Example:
  - 1 integer operation
    \[ \text{ALU} \text{op} \text{ dest, src1, src2} \]  # in 1 clock cycle
    In parallel with
  - 1 memory operation
    \[ \text{LD} \text{ dst, addr} \]  # in 2 clock cycles
    \[ \text{ST} \text{ src, addr} \]  # in 1 clock cycle
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2, r2, r3
3: ST r4, 4(r5)
4: LD r6, 8(r1)
5: ADD r6, r6, r2
6: ADD r6, r6, r4
7: ST r7, 0(r7)
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2,r2,r3
3: ST r4,4(r5)
4: LD r6,8(r1)
5: ADD r6,r6,r2
6: ADD r6,r6,r4
7: ST r7,0(r7)

\[ \text{READY} = \{1, 3, 4, 7\} \]
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2, r2, r3
3: ST r4, 4(r5)
4: LD r6, 8(r1)
5: ADD r6, r6, r2
6: ADD r6, r6, r4
7: ST r7, 0(r7)

READY= {1, 3, 4, 7}
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2, r2, r3
3: ST r4, 4(r5)
4: LD r6, 8(r1)
5: ADD r6, r6, r2
6: ADD r6, r6, r4
7: ST r7, 0(r7)

READY = {1, 3, 4, 7}
List Scheduling Example

1: LD  r2, 0(r1)
2: ADD r2, r2, r3
3: ST  r4, 4(r5)
4: LD  r6, 8(r1)
5: ADD r6, r6, r2
6: ADD r6, r6, r4
7: ST  r7, 0(r7)

READY= {2, 3, 4, 7}

<table>
<thead>
<tr>
<th>ALUop</th>
<th>MEM 1</th>
<th>MEM 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2,r2,r3
3: ST r4,4(r5)
4: LD r6,8(r1)
5: ADD r6,r6,r2
6: ADD r6,r6,r4
7: ST r7,0(r7)

READY= \{2, 3, 4, 7\}

<table>
<thead>
<tr>
<th>ALUop</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1</td>
</tr>
<tr>
<td>MEM 2</td>
<td>1</td>
</tr>
</tbody>
</table>
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2,r2,r3
3: ST r4,4(r5)
4: LD r6,8(r1)
5: ADD r6,r6,r2
6: ADD r6,r6,r4
7: ST r7,0(r7)

READY= \{3, 4, 7\}
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2,r2,r3
3: ST r4,4(r5)
4: LD r6,8(r1)
5: ADD r6,r6,r2
6: ADD r6,r6,r4
7: ST r7,0(r7)

READY = \{3, 4, 7\}
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2,r2,r3
3: ST r4,4(r5)
4: LD r6,8(r1)
5: ADD r6,r6,r2
6: ADD r6,r6,r4
7: ST r7,0(r7)

READY= \{3, 5, 7\}

<table>
<thead>
<tr>
<th>ALUop</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1 4</td>
</tr>
<tr>
<td>MEM 2</td>
<td>1 4</td>
</tr>
</tbody>
</table>
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2,r2,r3
3: ST r4,4(r5)
4: LD r6,8(r1)
5: ADD r6,r6,r2
6: ADD r6,r6,r4
7: ST r7,0(r7)

READY= \{3, 5, 7\}

<table>
<thead>
<tr>
<th>ALUop</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1</td>
</tr>
<tr>
<td>MEM 2</td>
<td>1</td>
</tr>
</tbody>
</table>
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2, r2, r3
3: ST r4, 4(r5)
4: LD r6, 8(r1)
5: ADD r6, r6, r2
6: ADD r6, r6, r4
7: ST r7, 0(r7)

READY = \{5, 7\}

<table>
<thead>
<tr>
<th>ALUop</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1 4 3</td>
</tr>
<tr>
<td>MEM 2</td>
<td>1 4</td>
</tr>
</tbody>
</table>
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2, r2, r3
3: ST r4, 4(r5)
4: LD r6, 8(r1)
5: ADD r6, r6, r2
6: ADD r6, r6, r4
7: ST r7, 0(r7)

READY= {5, 7}
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2, r2, r3
3: ST r4, 4(r5)
4: LD r6, 8(r1)
5: ADD r6, r6, r2
6: ADD r6, r6, r4
7: ST r7, 0(r7)

READY = \{6, 7\}

<table>
<thead>
<tr>
<th>ALUop</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MEM 2</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2,r2,r3
3: ST r4,4(r5)
4: LD r6,8(r1)
5: ADD r6,r6,r2
6: ADD r6,r6,r4
7: ST r7,0(r7)

\[
\text{READY} = \{6, 7\}
\]

<table>
<thead>
<tr>
<th>ALUop</th>
<th>2</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>MEM 2</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2,r2,r3
3: ST r4,4(r5)
4: LD r6,8(r1)
5: ADD r6,r6,r2
6: ADD r6,r6,r4
7: ST r7,0(r7)

READY = \{7\}

<table>
<thead>
<tr>
<th>ALUop</th>
<th>2</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>MEM 2</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2, r2, r3
3: ST r4, 4(r5)
4: LD r6, 8(r1)
5: ADD r6, r6, r2
6: ADD r6, r6, r4
7: ST r7, 0(r7)

READY = \{7\}

<table>
<thead>
<tr>
<th>ALUop</th>
<th>2</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>MEM 2</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
List Scheduling Example

1: LD r2, 0(r1)
2: ADD r2, r2, r3
3: ST r4, 4(r5)
4: LD r6, 8(r1)
5: ADD r6, r6, r2
6: ADD r6, r6, r4
7: ST r7, 0(r7)

READY = \{ \}

<table>
<thead>
<tr>
<th>ALUop</th>
<th>2</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>MEM 2</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Outline

- Overview of Instruction Scheduling
- List Scheduling
- Resource Constraints
- Interaction with Register Allocation
- Scheduling across Basic Blocks
- Trace Scheduling
- Scheduling for Loops
- Loop Unrolling
- Software Pipelining
Register Allocation and Instruction Scheduling

• If Register Allocation is before instruction scheduling
  – restricts the choices for scheduling
Example

1: LD r2, 0(r1)
2: ADD r3, r3, r2
3: LD r2, 4(r5)
4: ADD r6, r6, r2
Example

1: LD r2, 0(r1)
2: ADD r3, r3, r2
3: LD r2, 4(r5)
4: ADD r6, r6, r2
Example

1: LD  r2, 0(r1)
2:  ADD r3,r3,r2
3:  LD  r2,4(r5)
4:  ADD r6,r6,r2
Example

1: LD r2, 0(r1)
2: ADD r3, r3, r2
3: LD r2, 4(r5)
4: ADD r6, r6, r2

Anti-Dependence between 3 and 2.
There is really no data flowing...
How to “fix” this?
How about using a different Register?
Example

1: LD r2, 0(r1)  
2: ADD r3, r3, r2  
3: LD r4, 4(r5)  
4: ADD r6, r6, r4

Eliminated Anti-Dependence \textit{but}  
increased the number of registers  
i.e., increased Register Pressure
Example

1: LD r2, 0(r1)
2: ADD r3, r3, r2
3: LD r4, 4(r5)
4: ADD r6, r6, r4

<table>
<thead>
<tr>
<th>ALUop</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM 1</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MEM 2</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Register Allocation and Instruction Scheduling

• If Register Allocation is before Instruction Scheduling
  – restricts the choices for scheduling
Register Allocation and Instruction Scheduling

• If Register Allocation is before Instruction Scheduling
  – restricts the choices for scheduling

• If Instruction Scheduling before Register Allocation
  – Register allocation may spill registers
  – Will change the carefully done schedule!!!
Outline

• Overview of Instruction Scheduling
• List Scheduling
• Resource Constraints
• Interaction with Register Allocation
• Scheduling across Basic Blocks
• Trace Scheduling
• Scheduling for Loops
• Loop Unrolling
• Software Pipelining
Scheduling across Basic Blocks

• Number of Instructions in a Basic Block is small
  – Cannot keep a multiple units with long pipelines busy by just scheduling within a basic block

• Need to handle Control Dependence
  – Scheduling constraints across basic blocks
  – Scheduling policy
Moving across Basic Blocks

- Downward to adjacent Basic Block

```
A
B  C
```
Moving across Basic Blocks

- Downward to adjacent Basic Block
Moving across Basic Blocks

- Downward to adjacent Basic Block

- A path to B that does not execute A?
Moving across Basic Blocks

• Upward to adjacent Basic Block

A

B

C
Moving across Basic Blocks

- Upward to adjacent Basic Block

Diagram:
- Blocks A, B, and C connected as follows:
  - A connects to B and C
  - B connects only to A
  - C connects only to A
Moving across Basic Blocks

• Upward to adjacent Basic Block

B → A → C

• A path from C that does not reach A?
Outline

• Overview of Instruction Scheduling
• List Scheduling
• Resource Constraints
• Interaction with Register Allocation
• Scheduling across Basic Blocks
• Trace Scheduling
• Scheduling for Loops
• Loop Unrolling
• Software Pipelining
Trace Scheduling

- Find the most common Trace of Basic Blocks
  - Use profiling information
- Combine the Basic Blocks in the trace and schedule them as one Block
- Create clean-up code if the execution goes off-trace
Trace Scheduling
Trace Scheduling
Trace Scheduling

A
  ↓
B
  ↓
D
  ↓
E
  ↓
G
  ↓
H
Large Basic Blocks via Code Duplication

• Creating large extended Basic Blocks by duplication
Large Basic Blocks via Code Duplication

- Creating large extended Basic Blocks by duplication
- Schedule the larger Blocks
Scheduling Loops

• Loop bodies are small
• But, lot of time is spent in loops due to large number of iterations
• Need better ways to schedule loops
Loop Example

• Machine Model
  – One load/store unit
    • load 2 cycles
    • store 2 cycles
  – Two arithmetic units
    • add 2 cycles
    • branch 2 cycles (no delay slot)
    • multiply 3 cycles
  – Both units are pipelined (initiate one op each cycle)

• Source Code
  for i = 1 to N
Loop Example

- Source Code

```plaintext
for i = 1 to N
```

- Assembly Code

```plaintext
loop:
    ld    r6, (r2)
    mul   r6, r6, r3
    st    r6, (r2)
    add   r2, r2, 4
    ble   r2, r5, loop
```
Loop Example

• Assembly Code

```
loop:
  ld    r6, (r2)
  mul  r6, r6, r3
  st    r6, (r2)
  add  r2, r2, 4
  ble  r2, r5, loop
```

• Schedule (7 cycles per iteration) – excluding branch

```
<table>
<thead>
<tr>
<th>ld</th>
<th></th>
<th>st</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ld</td>
<td>st</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ble</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Outline

• Overview of Instruction Scheduling
• List Scheduling
• Resource Constraints
• Interaction with Register Allocation
• Scheduling across Basic Blocks
• Trace Scheduling
• Scheduling for Loops
• Loop Unrolling
• Software Pipelining
Loop Unrolling

• Unroll the Loop Body a few times

• Pros:
  – Create a much larger basic block for the body
  – Eliminate few loop bounds checks

• Cons:
  – Much larger program
  – Setup code (# of iterations < unroll factor)
  – beginning and end of the schedule can still have unused slots
Loop Example

```
loop:
  ld    r6, (r2)
  mul  r6, r6, r3
  st    r6, (r2)
  add  r2, r2, 4
  ble  r2, r5, loop
```
Loop Example

loop:
    ld r6, (r2)
    mul r6, r6, r3
    st r6, (r2)
    add r2, r2, 4
    ld r6, (r2)
    mul r6, r6, r3
    st r6, (r2)
    add r2, r2, 4
    ble r2, r5, loop
Loop Example

loop:

ld   r6, (r2)
mul  r6, r6, r3
st   r6, (r2)
add  r2, r2, 4
ld   r6, (r2)
mul  r6, r6, r3
st   r6, (r2)
add  r2, r2, 4
ble  r2, r5, loop

• Schedule (7 cycles per iteration)
Loop Unrolling

• Rename Registers
  – Use Different Registers in Different Loop Iterations
Loop Example

```
loop:
  ld   r6, (r2)
  mul r6, r6, r3
  st   r6, (r2)
  add r2, r2, 4
  ld   r6, (r2)
  mul r6, r6, r3
  st   r6, (r2)
  add r2, r2, 4
  ble  r2, r5, loop
```
Loop Example

```
loop:
  ld   r6, (r2)
  mul  r6, r6, r3
  st   r6, (r2)
  add  r2, r2, 4
  ld   r7, (r2)
  mul  r7, r7, r3
  st   r7, (r2)
  add  r2, r2, 4
  ble  r2, r5, loop
```
Loop Unrolling

• Rename Registers
  – Use Different Registers in Different Loop Iterations

• Eliminate Unnecessary Dependences
  – Use more registers to eliminate true, anti and output dependences
  – Eliminate dependent-chains of calculations when possible
Loop Example

```
loop:
    ld       r6, (r2)
    mul     r6, r6, r3
    st       r6, (r2)
    add     r2, r2, 4
    ld       r7, (r2)
    mul     r7, r7, r3
    st       r7, (r2)
    add     r2, r2, 4
    ble     r2, r5, loop
```
Loop Example

loop:
    ld      r6, (r1)
    mul     r6, r6, r3
    st      r6, (r1)
    add     r2, r1, 4
    ld      r7, (r2)
    mul     r7, r7, r3
    st      r7, (r2)
    add     r1, r2, 4
    ble     r1, r5, loop
Loop Example

loop:
  ld     r6, (r1)
  mul   r6, r6, r3
  st     r6, (r1)
  add   r2, r1, 4
  ld     r7, (r2)
  mul   r7, r7, r3
  st     r7, (r2)
  add   r1, r2, 4
  ble   r1, r5, loop
Loop Example

```
loop:
  ld    r6, (r1)
  mul  r6, r6, r3
  st    r6, (r1)
  add  r2, r1, 4
  ld    r7, (r2)
  mul  r7, r7, r3
  st    r7, (r2)
  add  r1, r1, 8
  ble  r1, r5, loop
```
Loop Example

```
loop:
  ld    r6, (r1)
mul   r6, r6, r3
st    r6, (r1)
add   r2, r1, 4
ld    r7, (r2)
mul   r7, r7, r3
st    r7, (r2)
add   r1, r1, 8
ble   r1, r5, loop
```

• Schedule (3.5 cycles per iteration)

```
<table>
<thead>
<tr>
<th>ld</th>
<th>ld</th>
<th>st</th>
<th>st</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ld</td>
<td>st</td>
<td>st</td>
</tr>
<tr>
<td>mul</td>
<td>mul</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>mul</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Outline

• Overview of Instruction Scheduling
• List Scheduling
• Resource Constraints
• Interaction with Register Allocation
• Scheduling across Basic Blocks
• Trace Scheduling
• Scheduling for Loops
• Loop Unrolling
• Software Pipelining
Software Pipelining

• Try to overlap Multiple Iterations so that the Slots will be filled

• Find the Steady-State Window so that:
  – All the instructions of the loop body are executed
  – But from different iterations
Loop Example

• Assembly Code

\[
\text{loop:} \\
\begin{align*}
\text{ld} & \quad r6, (r2) \\
\text{mul} & \quad r6, r6, r3 \\
\text{st} & \quad r6, (r2) \\
\text{add} & \quad r2, r2, 4 \\
\text{ble} & \quad r2, r5, \text{loop}
\end{align*}
\]

• Schedule

<table>
<thead>
<tr>
<th>Id</th>
<th>Id1</th>
<th>Id2</th>
<th>st</th>
<th>Id3</th>
<th>st1</th>
<th>Id4</th>
<th>Id5</th>
<th>Id6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Id1</td>
<td>Id2</td>
<td>st</td>
<td>Id3</td>
<td>st1</td>
<td>Id4</td>
<td>Id5</td>
<td>Id6</td>
</tr>
<tr>
<td>mul</td>
<td>mul1</td>
<td>mul2</td>
<td>ble</td>
<td>mul3</td>
<td>ble1</td>
<td>mul4</td>
<td>ble2</td>
<td>mul5</td>
</tr>
<tr>
<td>mul</td>
<td>mul1</td>
<td>mul2</td>
<td>ble</td>
<td>mul3</td>
<td>ble1</td>
<td>mul4</td>
<td>ble2</td>
<td>mul4</td>
</tr>
<tr>
<td>mul</td>
<td></td>
<td></td>
<td>add</td>
<td></td>
<td>add1</td>
<td>add2</td>
<td>add3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>add</td>
<td></td>
<td>add3</td>
</tr>
</tbody>
</table>

Pedro Diniz
pedro@isi.edu

Fall 2015
Loop Example

• Assembly Code

```assembly
loop:
    ld    r6, (r2)
    mul   r6, r6, r3
    st    r6, (r2)
    add   r2, r2, 4
    ble   r2, r5, loop
```

• Schedule

```
<table>
<thead>
<tr>
<th>Id</th>
<th>Id1</th>
<th>Id2</th>
<th>st</th>
<th>Id3</th>
<th>st1</th>
<th>Id4</th>
<th>Id5</th>
<th>Id6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Id1</td>
<td>Id2</td>
<td>st</td>
<td>Id3</td>
<td>st1</td>
<td>Id4</td>
<td>Id5</td>
<td>Id6</td>
</tr>
<tr>
<td>mul</td>
<td>mul1</td>
<td>mul2</td>
<td>ble</td>
<td>mul3</td>
<td>ble1</td>
<td>mul4</td>
<td>ble2</td>
<td>mul5</td>
</tr>
<tr>
<td>mul</td>
<td>mul1</td>
<td>mul2</td>
<td>ble</td>
<td>mul3</td>
<td>ble1</td>
<td>mul4</td>
<td>ble2</td>
<td>mul5</td>
</tr>
<tr>
<td>mul</td>
<td>mul1</td>
<td>mul2</td>
<td>ble</td>
<td>mul3</td>
<td>ble1</td>
<td>mul4</td>
<td>ble2</td>
<td>mul5</td>
</tr>
<tr>
<td>add</td>
<td>add1</td>
<td>add2</td>
<td>add3</td>
<td>add1</td>
<td>add2</td>
<td>add3</td>
<td>add3</td>
<td></td>
</tr>
</tbody>
</table>
```
Loop Example

• Assembly Code
  
  ```assembly
  loop:
     ld    r6, (r2)
  mul   r6, r6, r3
  st    r6, (r2)
  add   r2, r2, 4
  ble   r2, r5, loop
  ```

• Schedule (2 cycles per iteration)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld3</td>
<td>st1</td>
</tr>
<tr>
<td>st</td>
<td>ld3</td>
</tr>
<tr>
<td>mul2</td>
<td>ble</td>
</tr>
<tr>
<td>mul1</td>
<td>mul2</td>
</tr>
<tr>
<td>add</td>
<td>add1</td>
</tr>
</tbody>
</table>
Loop Example

- 4 Iterations are Overlapped
  - value of r3 and r5 don’t change
  - 4 regs for &A[i] (r2)
  - each address incremented by 4*4
  - 4 regs to keep value A[i] (r6)
  - Same registers can be reused after 4 of these blocks; generate code for 4 blocks, otherwise need to move

```plaintext
loop:
  ld    r6, (r2)
  mul   r6, r6, r3
  st    r6, (r2)
  add   r2, r2, 4
  ble   r2, r5, loop
```

```
<table>
<thead>
<tr>
<th></th>
<th>st1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld3</td>
<td></td>
</tr>
<tr>
<td>st</td>
<td>ld3</td>
</tr>
<tr>
<td>mul2</td>
<td></td>
</tr>
<tr>
<td>mul1</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
</tr>
</tbody>
</table>
```
Software Pipelining

- Optimal use of Resources
- Need a lot of Registers
  - Values in multiple iterations need to be kept separated

- Issues with Dependences:
  - Executing a store instruction in an iteration before branch instruction is executed for a previous iteration (writing when it should not have)
  - Loads and stores are issued out-of-order (need to figure-out dependencies before doing this)

- Code Generation Issues:
  - Generate pre-amble and post-amble code
  - Multiple blocks so no register copy is needed
Summary

• Overview of Instruction Scheduling
• List Scheduling
• Resource Constraints
• Interaction with Register Allocation
• Scheduling across Basic Blocks
• Trace Scheduling
• Scheduling for Loops
• Loop Unrolling
• Software Pipelining