Instruction Scheduling and Software Pipelining - 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline

- Instruction Scheduling
  - Simple Basic Block Scheduling
  - Trace, Superblock and Hyperblock scheduling
- Software pipelining
Basic block consists of micro-operation sequences (MOS), which are indivisible

Each MOS has several steps, each requiring resources

Each step of an MOS requires one cycle for execution

Precedence constraints and resource constraints must be satisfied by the scheduled program

PC’s relate to data dependences and execution delays
RC’s relate to limited availability of shared resources
The Basic Block Scheduling Problem

- Basic block is modelled as a digraph, \( G = (V, E) \)
  - \( R \): number of resources
  - Nodes (\( V \)): MOS; Edges (\( E \)): Precedence
  - Label on node \( v \)
    - resource usage functions, \( \rho_v(i) \) for each step of the MOS associated with \( v \)
    - length \( l(v) \) of node \( v \)
  - Label on edge \( e \): Execution delay of the MOS, \( d(e) \)

- Problem: Find the shortest schedule \( \sigma : V \rightarrow N \) such that
  \[ \forall e = (u, v) \in E, \; \sigma(v) - \sigma(u) \geq d(e) \]  
  \[ \forall i, \; \sum_{v \in V} \rho_v(i - \sigma(v)) \leq R, \text{ where} \]
  length of the schedule is \( \max_{v \in V} \{ \sigma(v) + l(v) \} \)
Instruction Scheduling - Precedence and Resource Constraints

Consider $R = 5$. Each MOS substep takes 1 time unit.

- At $i=4$, $\sigma_{v_4}(1) + \sigma_{v_3}(2) + \sigma_{v_2}(3) + \sigma_{v_1}(4) = 2 + 2 + 1 + 0 = 5 \leq R$, satisfied

- At $i=2$, $\sigma_{v_3}(0) + \sigma_{v_2}(1) + \sigma_{v_1}(2) = 3 + 3 + 2 = 8 > R$, NOT satisfied
A Simple List Scheduling Algorithm

Find the shortest schedule \( \sigma : V \rightarrow N \), such that precedence and resource constraints are satisfied. Holes are filled with NOPs.

FUNCTION ListSchedule (V,E)
BEGIN

Ready = root nodes of V; Schedule = \( \phi \);
WHILE Ready \( \neq \phi \) DO
BEGIN

v = highest priority node in Ready;
Lb = SatisfyPrecedenceConstraints (v, Schedule, \( \sigma \));
\( \sigma (v) = \) SatisfyResourceConstraints (v, Schedule, \( \sigma \), Lb);
Schedule = Schedule + \{v\};
Ready = Ready − \{v\} + \{u | NOT (u ∈ Schedule) AND \( \forall (w, u) \in E, w \in \) Schedule\};

END
RETURN \( \sigma \);
END
List Scheduling - Ready Queue Update

Already scheduled nodes: \( W \)  
Currently scheduled node: \( V \)  
Unscheduled nodes which will get into the Ready queue now: \( U \)  
Unscheduled nodes: \( X \)
FUNCTION SatisfyPrecedenceConstraint(v, Sched, σ)
BEGIN
    RETURN (max \( \sigma(u) + d(u, v) \))
END

FUNCTION SatisfyResourceConstraint(v, Sched, σ, Lb)
BEGIN
    FOR i := Lb TO \( \infty \) DO
        IF \( \forall 0 \leq j < l(v), \rho_v(j) + \sum_{u \in Sched} \rho_u(i + j - \sigma(u)) \leq R \) THEN
            RETURN (i);
        END
    END
END
Precedence Constraint Satisfaction

Lower bound for $\sigma(v) = 29$

Already scheduled nodes

Precedence constraint satisfaction:

$\nu$ can be scheduled only after all of $u_1$, $u_2$, and $u_3$, finish

Node to be scheduled

Lower bound for $\sigma(v)$

= \max(10+2, 25+4, 18+3)
= \max(12, 29, 21) = 29
Resource Constraint Satisfaction

Consider $R = 5$. Each MOS substep takes 1 time unit.

<table>
<thead>
<tr>
<th>MOS substeps (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>$\sigma(v_1)=0$</td>
</tr>
<tr>
<td>$\sigma(v_2)=1$</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>$\sigma(v_3)=4$</td>
</tr>
<tr>
<td>$\sigma(v_4)=5$</td>
</tr>
</tbody>
</table>

Schedule Time $\sigma(u)$

Time slots 2 and 3 are vacant because scheduling node $v_3$ in either of them violates resource constraints.
1. Height of the node in the DAG (i.e., longest path from the node to a terminal node)

2. \textit{Estart}, and \textit{Lstart}, the earliest and latest start times
   - Violating \textit{Estart} and \textit{Lstart} may result in pipeline stalls
   - \(Estart(v) = \max_{i=1,\ldots,k} (Estart(u_i) + d(u_i, v))\)
     where \(u_1, u_2, \ldots, u_k\) are predecessors of \(v\). \textit{Estart} value of the source node is 0.
   - \(Lstart(u) = \min_{i=1,\ldots,k} (Lstart(v_i) - d(u, v_i))\)
     where \(v_1, v_2, \ldots, v_k\) are successors of \(u\). \textit{Lstart} value of the sink node is set as its \textit{Estart} value.
   - \textit{Estart} and \textit{Lstart} values can be computed using a top-down and a bottom-up pass, respectively, either statically (before scheduling begins), or dynamically during scheduling.
Estart Computation

\[
\begin{align*}
E_{\text{start}}(v) &= \max (E_{\text{start}}(u_i) + d_i) \\
&= \max(25 + 4, 45 + 7, 16 + 2) \\
&= \max(29, 52, 18) = 52
\end{align*}
\]
Lstart Computation

\[ \text{Lstart}(v) = \min \left( \text{Lstart}(w_i) - d_i \right) \]
\[ \text{for } i = 4, \ldots, 6 \]
\[ = \min(12-2, 36-1, 21-3) \]
\[ = \min(10, 35, 18) = 10 \]
List Scheduling - Slack

1. A node with a lower $E_{\text{start}}$ (or $L_{\text{start}}$) value has a higher priority

2. $\text{Slack} = L_{\text{start}} - E_{\text{start}}$
   - Nodes with lower slack are given higher priority
   - Instructions on the critical path may have a slack value of zero and hence get priority
Simple List Scheduling - Example - 1

INSTRUCTION SCHEDULING - EXAMPLE

LEGEND

path length
node no.
exec time

latency

path length (n) = exec time (n), if n is a leaf

= max { latency (n,m) + path length (m) }

m ∈ succ (n)

Schedule = {3, 1, 2, 4, 6, 5}
latencies
- \textit{add}, \textit{sub}, \textit{store}: 1 cycle; \textit{load}: 2 cycles; \textit{mult}: 3 cycles

\textit{path length} and \textit{slack} are shown on the left side and right side of the pair of numbers in parentheses.

\begin{equation}
\begin{aligned}
c &= (a+4)+(a-2)*b; \\
b &= b+3;
\end{aligned}
\end{equation}

\begin{itemize}
    
    \item \textbf{High-Level Code}

    \begin{tabular}{|c|c|}
    \hline
    i1: & t1 \leftarrow \text{load } a \\
    i2: & t2 \leftarrow \text{load } b \\
    i3: & t3 \leftarrow t1 + 4 \\
    i4: & t4 \leftarrow t1 - 2 \\
    i5: & t5 \leftarrow t2 + 3 \\
    i6: & t6 \leftarrow t4 \times t2 \\
    i7: & t7 \leftarrow t3 + t6 \\
    i8: & c \leftarrow \text{st } t7 \\
    i9: & b \leftarrow \text{st } t5 \\
    \hline
    \end{tabular}

    \item \textbf{3-Address Code}

\end{itemize}

(c) DAG with \((\text{Estart}, \text{Lstart})\) Values
Simple List Scheduling - Example - 2 (contd.)

- Latencies
  - \textit{add, sub, store}: 1 cycle; \textit{load}: 2 cycles; \textit{mult}: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well
- Heuristic used: height of the node or slack

<table>
<thead>
<tr>
<th>int1</th>
<th>int2</th>
<th>mult</th>
<th>Cycle #</th>
<th>Instr.No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>i1, i2</td>
<td>(t_1 \leftarrow \text{load } a, t_2 \leftarrow \text{load } b)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>i4, i3</td>
<td>(t_4 \leftarrow t_1 - 2, t_3 \leftarrow t_1 + 4)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>i6, i5</td>
<td>(t_5 \leftarrow t_2 + 3, t_6 \leftarrow t_4 \times t_2)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td></td>
<td>i5 not sched. in cycle 2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td></td>
<td>due to shortage of \textit{int} units</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>i7</td>
<td>(t_7 \leftarrow t_3 + t_6)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>i8</td>
<td>(c \leftarrow \text{st } t_7)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>i9</td>
<td>(b \leftarrow \text{st } t_5)</td>
</tr>
</tbody>
</table>
Resource Usage Models -
Instruction Reservation Table

<table>
<thead>
<tr>
<th></th>
<th>$r_0$</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$t_1$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$t_3$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

No. of resources in the machine: 4
<table>
<thead>
<tr>
<th></th>
<th>$r_0$</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$\cdots$</th>
<th>$r_M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>$t_1$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$t_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$t_T$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M: No. of resources in the machine  
T: Length of the schedule
GRT is constructed as the schedule is built (cycle by cycle)
All entries of GRT are initialized to 0
GRT maintains the state of all the resources in the machine
GRTs can answer questions of the type:
“can an instruction of class I be scheduled in the current cycle (say $t_k$)?”
Answer is obtained by ANDing RT of I with the GRT starting from row $t_k$
  - If the resulting table contains only 0’s, then YES, otherwise NO
The GRT is updated after scheduling the instruction with a similar OR operation
Simple List Scheduling - Disadvantages

- Checking resource constraints is inefficient here because it involves repeated ANDing and ORing of bit matrices for many instructions in each scheduling step.
- Space overhead may become considerable, but still manageable.
Average size of a basic block is quite small (5 to 20 instructions)
- Effectiveness of instruction scheduling is limited
- This is a serious concern in architectures supporting greater ILP
  - VLIW architectures with several function units
  - superscalar architectures (multiple instruction issue)

Global scheduling is for a set of basic blocks
- Overlaps execution of successive basic blocks
- Trace scheduling, Superblock scheduling, Hyperblock scheduling, Software pipelining, etc.
Trace Scheduling

- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path)
- Identifying a trace
  - Identify the most frequently executed basic block
  - Extend the trace starting from this block, forward and backward, along most frequently executed edges
- Apply list scheduling on the trace (including the branch instructions)
- Execution time for the trace may reduce, but execution time for the other paths may increase
- However, overall performance will improve
Trace Example

for (i=0; i < 100; i++)
{
    if (A[i] == 0)
        B[i] = B[i] + s;
    else
        B[i] = A[i];
    sum = sum + B[i];
}

(a) High-Level Code

<table>
<thead>
<tr>
<th></th>
<th>% r1 ← 0</th>
<th>% r5 ← 0</th>
<th>% r6 ← 400</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>i1: r2 ← load a(r1)</td>
<td>i2: if (r2 != 0) goto i7</td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>i3: r3 ← load b(r1)</td>
<td>i4: r4 ← r3 + r7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i5: b(r1) ← r4</td>
<td>i6: goto i9</td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>i7: r4 ← r2</td>
<td>i8: b(r1) ← r2</td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>i9: r5 ← r5 + r4</td>
<td>i10: r1 ← r1 + 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i11: if (r1 &lt; r6) goto i1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Assembly Code

(c) Control Flow Graph
2-way issue architecture with 2 integer units
- add, sub, store: 1 cycle, load: 2 cycles, goto: no stall
- 9 cycles for the main trace and 6 cycles for the off-trace

<table>
<thead>
<tr>
<th>Time</th>
<th>Int. Unit 1</th>
<th>Int. Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i1: r2 ← load a(r1)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>i2: if (r2 != 0) goto i7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i3: r3 ← load b(r1)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i4: r4 ← r3 + r7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i5: b(r1) ← r4</td>
<td>i6: goto i9</td>
</tr>
<tr>
<td>5</td>
<td>i7: r4 ← r2</td>
<td>i8: b(r1) ← r2</td>
</tr>
<tr>
<td>6</td>
<td>i9: r5 ← r5 + r4</td>
<td>i10: r1 ← r1 + 4</td>
</tr>
<tr>
<td>7 (4)</td>
<td>i10: r1 ← r1 + 4</td>
<td></td>
</tr>
<tr>
<td>8 (5)</td>
<td>i11: if (r1 &lt; r6) goto i1</td>
<td></td>
</tr>
</tbody>
</table>
Trace Scheduling: Example

(i1) load r2, a(i1)
(i2) bnez r2, i7

(i3) load r3, b(r1)
(i4) add r4, r3, r7
(i5) st b(r1), r4
(i6) br i9

(i7) mov r4, r2
(i8) st b(r1), r2

(i9) add r5, r5, r4
(i10) add r1, r1, 4
(i11) breq r1, r6, il
6 cycles for the main trace and 7 cycles for the off-trace

<table>
<thead>
<tr>
<th>Time</th>
<th>Int. Unit 1</th>
<th>Int. Unit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i1: r2 ← load a(r1)</td>
<td>i3: r3 ← load b(r1)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i2: if (r2 != 0) goto i7</td>
<td>i4: r4 ← r3 + r7</td>
</tr>
<tr>
<td>3</td>
<td>i5: b(r1) ← r4</td>
<td></td>
</tr>
<tr>
<td>4 (5)</td>
<td>i9: r5 ← r5 + r4</td>
<td>i10: r1 ← r1 + 4</td>
</tr>
<tr>
<td>5 (6)</td>
<td>i11: if (r1 &lt; r6) goto i1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i7: r4 ← r2</td>
<td>i8: b(r1) ← r2</td>
</tr>
<tr>
<td>4</td>
<td>i12: goto i9</td>
<td></td>
</tr>
</tbody>
</table>
Trace Scheduling - Issues

- *Side exits* and *side entrances* are ignored during scheduling of a trace.
- Required compensation code is inserted during book-keeping (after scheduling the trace).
- Speculative code motion - *load* instruction moved ahead of conditional branch:
  - Example: Register r3 should not be live in block B3 (off-trace path).
  - May cause unwanted exceptions.
    - Requires additional hardware support!
What compensation code is required when Instr 1 is moved below the side exit in the trace?
Compensation Code (contd.)

<table>
<thead>
<tr>
<th>\vdots</th>
<th>\vdots</th>
<th>\vdots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr 1</td>
<td>Instr 2</td>
<td></td>
</tr>
<tr>
<td>Instr 2</td>
<td>Instr 3</td>
<td></td>
</tr>
<tr>
<td>Instr 3</td>
<td>Instr 4</td>
<td></td>
</tr>
<tr>
<td>Instr 4</td>
<td>Instr 1</td>
<td></td>
</tr>
<tr>
<td>Instr 5</td>
<td>Instr 5</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{Instr 1} \]
What compensation code is required when Instr 5 moves above the side entrance in the trace?
Compensation Code (contd.)

```
  . .
  . .
Instr 1
Instr 2
Instr 3
Instr 4
Instr 5
  . .
  . .
```

```
  . .
  . .
Instr 1
Instr 5
Instr 5
Instr 2
Instr 3
Instr 4
  . .
  . .
```

Instr 5