Instruction Scheduling and Software Pipelining - 3

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
Global Acyclic Scheduling

- 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
Superblock Scheduling

- A Superblock is a trace without side entrances
  - Control can enter only from the top
  - Many exits are possible
  - Eliminates several book-keeping overheads

- Superblock formation
  - Trace formation as before
  - Tail duplication to avoid side entrances into a superblock
  - Code size increases
Superblock Example

- 5 cycles for the main trace and 6 cycles for the off-trace

(a) Control Flow Graph

(b) Superblock Schedule

<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>i10: r1 ← r1 + 4</td>
</tr>
<tr>
<td>4</td>
<td>i9: r5 ← r5 + r4</td>
<td>i11: if (r1&lt;r6) goto i1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i7: r4 ← r2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i9’: r5 ← r5 + r4</td>
<td>i8: b(r1) ← r2</td>
</tr>
<tr>
<td>5</td>
<td>i11’: if (r1&lt;r6) goto i1</td>
<td>i10’: r1 ← r1 + 4</td>
</tr>
</tbody>
</table>

Y.N. Srikant Instruction Scheduling
Hyperblock Scheduling

- Superblock scheduling does not work well with control-intensive programs which have many control flow paths.
- Hyperblock scheduling was proposed to handle such programs.
- Here, the control flow graph is IF-converted to eliminate conditional branches.
- IF-conversion replaces conditional branches with appropriate predicated instructions.
- Now, control dependence is changed to a data dependence.
IF-Conversion Example

for $l = 1$ to $100$ do 
  if $(A(l) <= 0)$ then continue
  $A(l) = B(l) + 3$

for $l = 1$ to $N$ do 
  $S1: A(l) = D(l) + 1$
  $S2: \text{if } (B(l) > 0) \text{ then}$
  $S3: C(l) = C(l) + A(l)$
  $S4: \text{else } D(l+1) = D(l+1) + 1$
end if

for $l = 1$ to $100$ do 
  $p = (A(l) <= 0)$
  $(\neg p)$ $A(l) = B(l) + 3$

for $l = 1$ to $N$ do 
  $S1: A(l) = D(l) + 1$
  $S2: p = (B(l) > 0)$
  $S3: (p) C(l) = C(l) + A(l)$
  $S4: (\neg p) D(l+1) = D(l+1) + 1$
end if
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

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

(b) Assembly Code

(c) Control Flow Graph
Hyperblock Example

- 6 cycles for the entire set of predicated instructions
- Instructions i3 and i4 can be executed speculatively and can be moved up, instead of being scheduled after cycle 2

(a) Control Flow Graph

(b) Hyperblock Schedule

<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>i2’: p1 ← (r2 == 0)</td>
<td>i4: r4 ← r3 + r7</td>
</tr>
<tr>
<td>2</td>
<td>i5: b(r1) ← r4, if p1</td>
<td>i8: b(r1) ← r2, if !p1</td>
</tr>
<tr>
<td>3</td>
<td>i10: r1 ← r1 + 4</td>
<td>i7: r4 ← r2, if !p1</td>
</tr>
<tr>
<td>4</td>
<td>i9: r5 ← r5 + r4</td>
<td>i11: if (r1&lt;r6) goto i1</td>
</tr>
</tbody>
</table>
Introduction to Software Pipelining

- Overlaps execution of instructions from multiple iterations of a loop
- Executes instructions from different iterations in the same pipeline, so that pipelines are kept busy without stalls
- Objective is to sustain a high initiation rate
  - Initiation of a subsequent iteration may start even before the previous iteration is complete
- Unrolling loops several times and performing global scheduling on the unrolled loop
  - Exploits greater ILP within unrolled iterations
  - Very little or no overlap across iterations of the loop
More complex than instruction scheduling

NP-Complete

Involves finding initiation interval for successive iterations

- Trial and error procedure
- Start with minimum II, schedule the body of the loop using one of the approaches below and check if schedule length is within bounds
  - Stop, if yes
  - Try next value of II, if no

Requires a modulo reservation table (GRT with II columns and R rows)

Schedule lengths are dependent on II, dependence distance between instructions and resource contentions
Software Pipelining Example-1

```
for (i=1; i<=n; i++) {
    a[i+1] = a[i] + 1;
    b[i] = a[i+1]/2;
    c[i] = b[i] + 3;
    d[i] = c[i]
}
```

```
1       S1
T 2      S2  S1
3        S3  S2  S1
I 4      S4  S3  S2  S1
5        S4  S3  S2  S1
M 6      S4  S3  S2  S1
7        S4  S3  S2  S1
E 8      S4  S3  S2  S1
9        S4  S3  S4
10       S4
```

(dep.dist, delay)
No. of tokens present on an arc indicates the dependence distance

```
for (i = 0; i < n; i++) {
    a[i] = s * a[i];
}
```

(a) High-Level Code

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i0</td>
<td>i1</td>
<td>i2</td>
<td>i3</td>
<td>i4</td>
</tr>
<tr>
<td>t3</td>
<td>t4</td>
<td>a(t0)</td>
<td>t0</td>
<td>t1</td>
</tr>
<tr>
<td></td>
<td>load a(t0)</td>
<td>t2 * t3</td>
<td>t0 + 4</td>
<td>t1 - 1</td>
</tr>
</tbody>
</table>

(b) Instruction Sequence

Y.N. Srikant  Software Pipelining Example
Number of tokens present on an arc indicates the dependence distance

Assume that the possible dependence from i2 to i0 can be disambiguated

Assume 2 INT units (latency 1 cycle), 2 FP units (latency 2 cycles), and 1 LD/STR unit (latency 2 cycles/1 cycle)

Branch can be executed by INT units

Acyclic schedule takes 5 cycles (see figure)

Corresponds to an initiation rate of 1/5 iteration per cycle

Cyclic schedule takes 2 cycles (see figure)
### Acyclic Schedule

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i0: load</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i1: mult, i3: add, i4: sub</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i2: store, i5: bge</td>
</tr>
</tbody>
</table>

### Cyclic Schedule

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>i4: sub</td>
</tr>
<tr>
<td></td>
<td>i1: mult</td>
</tr>
<tr>
<td></td>
<td>i0: load</td>
</tr>
<tr>
<td>5</td>
<td>i2: store</td>
</tr>
<tr>
<td></td>
<td>i5: bge</td>
</tr>
<tr>
<td></td>
<td>i3: add</td>
</tr>
<tr>
<td>Time Step</td>
<td>Iter. 0</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>0</td>
<td>i0 : ld</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i1 : mult</td>
</tr>
<tr>
<td>3</td>
<td>i3 : add</td>
</tr>
<tr>
<td>4</td>
<td>i4 : sub</td>
</tr>
<tr>
<td>5</td>
<td>i2 : st</td>
</tr>
<tr>
<td></td>
<td>i5 : bge</td>
</tr>
<tr>
<td>6</td>
<td>i4 : sub</td>
</tr>
<tr>
<td>7</td>
<td>i2 : st</td>
</tr>
<tr>
<td></td>
<td>i5 : bge</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

A Software Pipelined Schedule with II = 2
for $i = 1$ to $n$ {
    0: $t_0[i] = a[i] + b[i]$;
    1: $t_1[i] = c[i] \times \text{const1}$;
    2: $t_2[i] = d[i] + e[i-2]$;
    3: $t_3[i] = t_0[i] + c[i]$;
    4: $t_4[i] = t_1[i] + t_2[i]$;
    5: $e[i] = t_3[i] \times t_4[i]$;
}

Dependence Graph

Pipe stages

Loop unrolled to reveal the software pipeline

2 multipliers, 2 adders, 1 cluster, single cycle operations
Automatic Parallelization - 1

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Automatic conversion of sequential programs to parallel programs by a compiler

Target may be a vector processor (vectorization), a multi-core processor (concurrentization), or a cluster of loosely coupled distributed memory processors (parallelization)

Parallelism extraction process is normally a source-to-source transformation

Requires dependence analysis to determine the dependence between statements

Implementation of available parallelism is also a challenge
- For example, can all the iterations of a 2-nested loop be run in parallel?
for \( I = 1 \) to 100 do {
    \( X(I) = X(I) + Y(I) \)
}

can be converted to

\[
X(1:100) = X(1:100) + Y(1:100)
\]

The above code can be run on a vector processor in \( O(1) \) time. The vectors \( X \) and \( Y \) are fetched first and then the vector \( X \) is written into
for \( I = 1 \) to 100 do {
    \( X(I) = X(I) + Y(I) \)
}

can be converted to

forall \( I = 1 \) to 100 do {
    \( X(I) = X(I) + Y(I) \)

The above code can be run on a multi-core processor with all the 100 iterations running as separate threads. Each thread “owns” a different \( I \) value
for $I = 1$ to 100 do {
    $X(I+1) = X(I) + Y(I)$
}

cannot be converted to

\[ X(2:101) = X(1:100) + Y(1:100) \]

because of dependence as shown below

\[
\begin{align*}
X(2) &= X(1) + Y(1) \\
X(3) &= X(2) + Y(2) \\
X(4) &= X(3) + Y(3) \\
odot
\end{align*}
\]
Data Dependence Relations

Flow or true dependence

\[ \text{S1: } X = \ldots \]
\[ \text{S2: } \ldots = X \]

\[ \delta \]

Anti-dependence

\[ \text{S1: } \ldots = X \]
\[ \text{S2: } X = \ldots \]

\[ \delta \]

Output dependence

\[ \text{S1: } X = \ldots \]
\[ \text{S2: } X = \ldots \]

\[ \delta^o \]
Data Dependence Direction Vector

- Data dependence relations are augmented with a direction of data dependence (direction vector).
- There is one direction vector component for each loop in a nest of loops.
- The *data dependence direction vector* (or direction vector) is \( \Psi = (\Psi_1, \Psi_2, \ldots, \Psi_d) \), where \( \Psi_k \in \{<, =, >, \leq, \geq, \neq, *\} \).
- Forward or “<” direction means dependence from iteration \( i \) to \( i + k \) (*i.e.*, computed in iteration \( i \) and used in iteration \( i + k \)).
- Backward or “>” direction means dependence from iteration \( i \) to \( i - k \) (*i.e.*, computed in iteration \( i \) and used in iteration \( i - k \)). This is not possible in single loops and possible in two or higher levels of nesting.
- Equal or “=” direction means that dependence is in the same iteration (*i.e.*, computed in iteration \( i \) and used in iteration \( i \)).
Direction Vector Example 1

for J = 1 to 100 do {
  S: \( X(J) = X(J) + c \)
}

S \( \delta_\leq \) S
\[
\begin{align*}
X(1) &= X(1) + c \\
X(2) &= X(2) + c
\end{align*}
\]

for J = 1 to 99 do {
  S: \( X(J+1) = X(J) + c \)
}

S \( \delta_\leq \) S
\[
\begin{align*}
X(2) &= X(1) + c \\
X(3) &= X(2) + c
\end{align*}
\]

for J = 1 to 99 do {
  S: \( X(J) = X(J+1) + c \)
}

S \( \delta_\leq \) S
\[
\begin{align*}
X(1) &= X(2) + c \\
X(2) &= X(3) + c
\end{align*}
\]

for J = 99 downto 1 do {
  S: \( X(J) = X(J+1) + c \)
}

S \( \delta_\leq \) S
\[
\begin{align*}
X(99) &= X(100) + c \\
X(98) &= X(99) + c
\end{align*}
\]

note ‘-ve’ increment

for J = 2 to 101 do {
  S: \( X(J) = X(J-1) + c \)
}

S \( \delta_\leq \) S
\[
\begin{align*}
X(2) &= X(1) + c \\
X(3) &= X(2) + c
\end{align*}
\]