Software Pipelining

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design
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
Iterative modulo scheduling
- Similar to list scheduling, computes priorities and uses operation scheduling (details later)
- Uses Modulo Reservation Tables (MRT)
  - A global resource reservation table with \( \Pi \) columns and \( R \) rows
  - MRT records resource usage of the schedule (of the kernel) as it is constructed
  - Initially all entries are 0
  - If an instruction uses a resource \( r \) at time step \( t \), then the entry \( MRT(r, t \mod \Pi) \) is set to 1

Slack scheduling
- Uses earliest and latest issue times for each instruction (difference is slack)
- Schedules an instruction within its slack
- Also uses MRT
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

Schedule lengths are dependent on II, dependence distance between instructions and resource contentions
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]
}
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>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>t1</td>
<td>t2</td>
<td>a(t0)</td>
<td>t3</td>
<td>t4</td>
<td>i0: load a(t0)</td>
</tr>
<tr>
<td>% 0</td>
<td>% (n-1)</td>
<td>% s</td>
<td>t4</td>
<td>t2 * t3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>t0</td>
<td>t0 + 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i1: t4</td>
<td>t1 - 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2: a(t0)</td>
<td>t4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i3: t0</td>
<td>t1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i4: t1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i5: if (t1 ≥ 0) goto i0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Instruction Sequence

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**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>
</tbody>
</table>
A Software Pipelined Schedule with \( II = 2 \)
for i = 1 to n {
    0: t0[i] = a[i] + b[i];
    1: t1[i] = c[i] * const1;
    2: t2[i] = d[i] + e[i-2];
    3: t3[i] = t0[i] + c[i];
    4: t4[i] = t1[i] + t2[i];
    5: e[i] = t3[i] * t4[i];
}

Dependence Graph

Pipe stages

2 multipliers, 2 adders, 1 cluster, single cycle operations

Loop unrolled to reveal the software pipeline
Minimum Initiation Interval (MII)

- Minimum time before which, successive iterations cannot be started
- $$MII = \max(ResMII, RecMII)$$
  - $$ResMII$$ is the minimum MII due to resource constraints
  - $$RecMII$$ is the minimum MII due to recurrences or cyclic data dependences
Resource Minimum Initiation Interval ($ResMII$)

- Very expensive to determine exactly
- For pipelined function units

\[ ResMII = \max_{r} \left \lceil \frac{N_r}{F_r} \right \rceil \]  \hspace{1cm} (1)

where $N_r$ represents the number of instructions that execute on a functional unit of type $r$, and $F_r$ is the number of functional units of type $r$.

- For non-pipelined FUs or FUs with complex structural hazards

\[ ResMII = \max_{r} \left \lceil \sum_a \frac{N_{a,r}}{F_r} \right \rceil \]  \hspace{1cm} (2)

where $N_{a,r}$ represents the maximum number of time steps for which instruction $a$ uses any of the stages of a functional unit of type $r$. For example, for a non-pipelined FU, $N_{a,r}$ equals to the latency of the functional unit.
Resource MII Example - Fully Pipelined FU

\[ ResMII = \max(ResMII_{\text{INT}}, ResMII_{\text{FP}}, ResMII_{\text{LD/STR}}) \]  

\[ ResMII = \max \left( \frac{3}{2}, \frac{1}{2}, \frac{2}{1} \right) = 2 \]

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

(a) High-Level Code

(b) Instruction Sequence

(c) Dependence graph

Software Pipelining Example
Resource MII Example 2

<table>
<thead>
<tr>
<th>Resources</th>
<th>INT function unit</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>r₀</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>r₁</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>r₂</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resources</th>
<th>LD/ST function unit</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>r₀</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>r₁</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>r₂</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resources</th>
<th>FP function unit</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>r₀</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>r₁</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>r₂</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

- i₀: r₀(2), r₁(2); i₁: r₂(3)
- i₂: r₀(2), r₁(2); i₃: r₀(2), r₁(2)
- i₄: r₀(2), r₁(2); i₅: r₀(2), r₁(2)

Resources: r₀(8), r₁(8), r₂(6)

ResMII = max (r₀:10/8, r₁:10/8, r₂:3/6) = max (1.25, 1.25, 0.5) = 2
Recurrence MII

- Recurrence Minimum Initiation Interval (RecMII)
  - Dependent on the cycle length (both delay length and distance length) in the dependence graph
  
  \[ RecMII = \max_{c \in \text{cycles}} \left\lceil \frac{\text{delay}(c)}{\text{distance}(c)} \right\rceil \]
  
  - Can be computed by enumerating all cycles
Recurrence MII Example

\[ \text{RecMII} = \max(\text{RecMII}_{\text{cycle on } i_3}, \text{RecMII}_{\text{cycle on } i_4}) \]  \tag{5} \\
\[ \text{RecMII} = \max \left( \frac{1}{1}, \frac{1}{1} \right) = 1 \]  \tag{6} 

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

(a) High-Level Code

<table>
<thead>
<tr>
<th></th>
<th>% t0 ← 0 %</th>
<th>% t1 ← (n-1) %</th>
<th>% t2 ← s %</th>
</tr>
</thead>
<tbody>
<tr>
<td>i0:</td>
<td>t3 ← load a(t0)</td>
<td>t4 ← t2 * t3</td>
<td>a(t0) ← t4</td>
</tr>
<tr>
<td>i1:</td>
<td>t0 ← t0 + 4</td>
<td>t1 ← t1 - 1</td>
<td></td>
</tr>
<tr>
<td>i2:</td>
<td>if (t1 ≥ 0) goto i0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Instruction Sequence

Software Pipelining Example
ResMII \(=\) \(\max\left(\frac{4}{2}, \frac{2}{2}\right) = 2 \) \tag{7}

RecMII \(=\) \(\max\left(\left\lceil\frac{1 + 1 + 1}{0 + 0 + 2}\right\rceil\right) = \left\lceil\frac{3}{2}\right\rceil = 2 \) \tag{8}

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: \(e[i] = t_3[i] \times t_4[i];\)
}\n
Program

Dependence Graph

Pipe stages

Loop unrolled to reveal the software pipeline

2 multipliers, 2 adders, 1 cluster, single cycle operations

Y.N. Srikant  Software Pipelining
Modulo Scheduling Algorithm

1. Compute $MII$ and set $II$ to $MII$
2. Compute priority for each node
   - Height of a node is one of the priority functions and is described later
   - Height is computed using both delay and distance
3. Choose an operation of highest priority for scheduling
4. Compute $E_{start}$ for the operation (described later)
5. Try slots within the range $(E_{start}, E_{start} + II - 1)$, for resource contentions (all ranges are modulo $II$)
Modulo Scheduling Algorithm

6 If one is available, then schedule the instruction; this may involve unscheduling those immediate successors of the instruction, with whom there is a dependence conflict (no resource conflicts are possible; this has just been checked before scheduling the instruction)

7 If none is available
   - choose $E_{\text{start}}$, if the instruction has not been scheduled so far
   - choose $prev\text{-sched}\text{-time}+1$ if the instruction was previously scheduled at $prev\text{-sched}\text{-time}$
   - this will invariably involve unscheduling all the instructions which have resource contentions with the instruction being scheduled

8 If there have been too many failures of the above types (6) or (7), then increment II and repeat the steps
Operation Scheduling

- Ready list has no use here because unscheduling of previously scheduled instructions is possible
- MRT with II columns and R rows is used to record commitments of scheduled instructions
- Conflict at time $T$ means conflict at $T + k \times II$ and $T - k \times II$

\[
E_{\text{start}}(P) = \max_{Q \in \text{Pred}(P)} \begin{cases} 
0, & \text{if } Q \text{ is unscheduled} \\
\max(0, \text{SchedTime}(Q) + \text{Delay}(Q, P) - II \times \text{Distance}(Q, P)), & \text{otherwise}
\end{cases}
\]

\[
\text{Height}(P) = \begin{cases} 
0, & \text{if } P \text{ is the STOP pseudo-op} \\
\max_{Q \in \text{Succ}(P)} (\text{Height}(Q) + \text{Delay}(P, Q) - II \times \text{Distance}(P, Q)), & \text{otherwise}
\end{cases}
\]

- Note that only scheduled predecessors will be considered in the computation of $E_{\text{start}}$
Instances of a single variable defined in a loop are active simultaneously in different concurrently active iterations (see figure in next slide):
- Value produced by $i_1$ in time step 2 is used by $i_2$ only in time step 5
- However, another instance of $i_1$ from \textit{iter 1} in time step 4 could overwrite the destination register
- Assigning the same register for each such variable will be incorrect

Automatic register renaming through rotating register sets is one hardware solution.

Unrolling the loop as many as $II$ times (max) and then applying the usual RA is another solution (Modulo-variable expansion):
- This process essentially renames the destination registers appropriately
- Increases $II$
## Interacting Live Range Problem

A Software Pipelined Schedule with II = 2

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Iter. 0</th>
<th>Iter. 1</th>
<th>Iter. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i0 : ld</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>i1 : mult</td>
<td>i0 : ld</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>i3 : add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>i4 : sub</td>
<td>i1 : mult</td>
<td>i0 : ld</td>
</tr>
<tr>
<td>5</td>
<td>i2 : st</td>
<td>i3 : add</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i5 : bge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>i4 : sub</td>
<td>i1 : mult</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>i2 : st</td>
<td>i3 : add</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i5 : bge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>i4 : sub</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>i2 : st</td>
<td>i5 : bge</td>
</tr>
</tbody>
</table>
Register requirement is higher than the available no. of registers

- Spill a few variables to memory
- Register spills need additional loads and stores
- If the memory unit is saturated in the kernel, and additional LD/STR cannot be scheduled
  - II value needs to be increased and loop must be rescheduled
- Reschedule loop with a larger II but without inserting spills
  - Increased II in general reduces register requirement of the schedule
- Generally, increasing II produces worse schedules than adding spill code
Handling Loops With Multiple Basic Blocks

- Hierarchical reduction
  - Two branches of a conditional are first scheduled independently
  - Entire conditional is then treated as a single node
    - Resource requirements is union of the resource requirements of the two branches
    - Length of schedule (latency) equal to the max of the lengths of the branches
  - After the entire loop is scheduled, conditionals are reinserted
- IF-Conversion and then scheduling the predicated code (resource usage here is the sum of the usages of the two branches)