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

Approaches to Software Pipelining

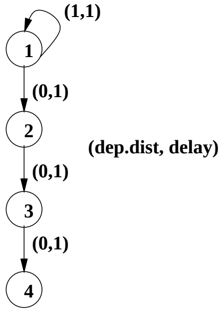
- Iterative modulo scheduling
  - Similar to list scheduling, computes priorities and uses operation scheduling (details later)
  - Uses Modulo Reservation Tables (MRT)
    - A global resource reservation table with II columns and R rows
    - MRT records resource usage of the schedule (of the kernel)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

Introduction to Software Pipelining - contd.

- 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

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]
}
```



		Iterations						
	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
	9						S4	S3
	10							S4

Software Pipelining Example-2.1

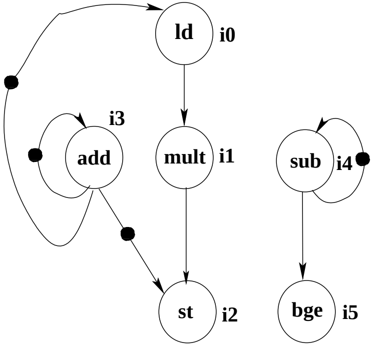
No. of tokens present on an arc indicates the dependence distance

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

(a) High-Level Code

```
     % t0 ← 0 %
     % t1 ← (n-1) %
     % t2 ← s %
i0:  t3     ←  load a(t0)
i1:  t4     ←  t2 * t3
i2:  a(t0)  ←  t4
i3:  t0     ←  t0 + 4
i4:  t1     ←  t1 - 1
i5:  if (t1 ≥ 0) goto i0
```

(b) Instruction Sequence



(c) Dependence graph

Software Pipelining Example

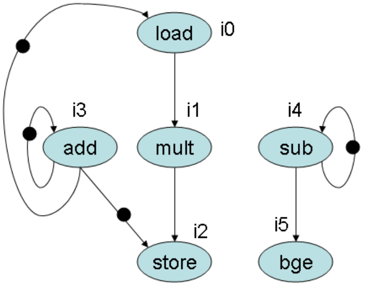
Software Pipelining Example-2.2

- 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 and Cyclic Schedules

Acyclic Schedule

0	i0: load
1	
2	i1: mult, i3: add, i4: sub
3	
4	i2: store, i5: bge



Cyclic Schedule

4	i4: sub	i1: mult	i0: load
5	i2: store i5: bge	i3: add	

Software Pipelining Example-2.3

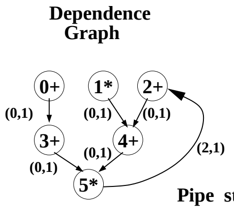
Time Step	Iter. 0	Iter. 1	Iter. 2	
0	i0 : ld			
1				Prolog
2	i1 : mult	i0 : ld		
3	i3 : add			
4	i4 : sub	i1 : mult	i0 : ld	Kernel
5	i2 : st i5 : bge	i3 : add		
6		i4 : sub	i1 : mult	
7		i2 : st i5 : bge	i3 : add	Epilog
8			i4 : sub	
9			i2 : st i5 : bge	

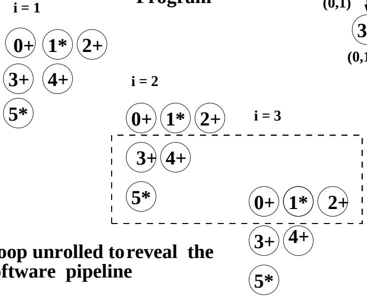
A Software Pipelined Schedule with II = 2

Software Pipelining Example-3

```
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];
}
```

Program





Loop unrolled to reveal the software pipeline

Pipe stages

time	PS0	PS1
0	3+ 4+	
1	5*	0+ 1* 2+

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

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

$$ResMII = \max_{\forall\, r} \left(\left\lceil \frac{N_r}{F_r} \right\rceil \right) \qquad (1)$$

  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

$$ResMII = \max_{\forall\, r} \left\lceil \frac{\sum_a N_{a,r}}{F_r} \right\rceil \qquad (2)$$

  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

$$ResMII = \max(ResMII_{INT}, ResMII_{FP}, ResMII_{LD/STR}) \quad (3)$$

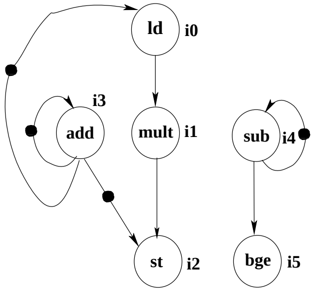
$$ResMII = \max\left(\frac{3}{2}, \frac{1}{2}, \frac{2}{1}\right) = 2 \quad (4)$$

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

(a) High-Level Code

	% t0 ← 0 %
	% t1 ← (n-1) %
	% t2 ← s %
i0:	t3 ← load a(t0)
i1:	t4 ← t2 * t3
i2:	a(t0) ← t4
i3:	t0 ← t0 + 4
i4:	t1 ← t1 - 1
i5:	if (t1 ≥ 0) goto i0

(b) Instruction Sequence



(c) Dependence graph

Software Pipelining Example

Resource MII Example 2

Time

Resources	0	1	2
r_0	1	1	0
r_1	0	1	1
r_2	0	0	0

INT function unit

Time

Resources	0	1	2
r_0	0	1	1
r_1	1	1	0
r_2	0	0	0

LD/ST function unit

Time

Resources	0	1	2
r_0	1	1	0
r_1	0	1	1
r_2	0	0	0

FP function unit

i_0: $r_0(2)$, $r_1(2)$; i_1: $r_2(3)$
i_2: $r_0(2)$, $r_1(2)$; i_3: $r_0(2)$, $r_1(2)$
i_4: $r_0(2)$, $r_1(2)$; i_5: $r_0(2)$, $r_1(2)$

Resources: $r_0(8)$, $r_1(8)$, $r_2(6)$

ResMII = max (r_0:10/8, r_1:10/8, r_2:3/6)
= 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
  - $RecMII = \max_{c \in cycles} \left\lceil \frac{delay(c)}{distance(c)} \right\rceil$
  - Can be computed by enumerating all cycles

Recurrence MII Example

$$RecMII = \max(RecMII_{cycle\ on\ i3}, RecMII_{cycle\ on\ i4}) \tag{5}$$

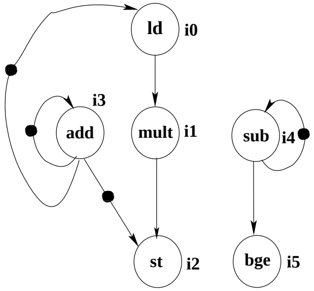
$$RecMII = \max\left(\frac{1}{1}, \frac{1}{1}\right) = 1 \tag{6}$$

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

(a) High-Level Code

	% t0 ← 0 %
	% t1 ← (n-1) %
	% t2 ← s %
i0:	t3 ← load a(t0)
i1:	t4 ← t2 * t3
i2:	a(t0) ← t4
i3:	t0 ← t0 + 4
i4:	t1 ← t1 - 1
i5:	if (t1 ≥ 0) goto i0

(b) Instruction Sequence



(c) Dependence graph

Software Pipelining Example

ResMII and RecMII Example - Fully Pipelined FUs

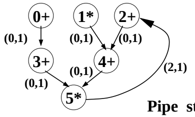
$$ResMII = \max\left(\frac{4}{2}, \frac{2}{2}\right) = 2 \tag{7}$$

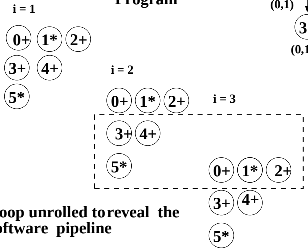
$$RecMII = \max\left(\left\lceil\frac{1+1+1}{0+0+2}\right\rceil\right) = \left(\left\lceil\frac{3}{2}\right\rceil\right) = 2 \tag{8}$$

```
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];
 }
```

Program

Dependence Graph





Loop unrolled to reveal the software pipeline

Pipe stages

time	PS0	PS1
0	3+ 4+	
1	5*	0+ 1* 2+

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

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 *Estart* for the operation (described later)
5. Try slots within the range (*Estart*, *Estart*+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 *Estart*, if the instruction has not been scheduled so far
   - choose *prev-sched-time*+1 if the instruction was previously scheduled at *prev-sched-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 $T + k * II$ and $T - k * II$

$$Estart(P) = \max_{Q \in Pred(P)} \begin{cases} 0, \ if\ Q\ is\ unscheduled \\ max(0, SchedTime(Q) + Delay(Q,P) \\ \quad -II * Distance(Q,P)), \ otherwise \end{cases}$$

$$Height(P) = \begin{cases} 0, \ if\ P\ is\ the\ STOP\ pseudo-op \\ \max_{Q \in Succ(P)} (Height(Q) + Delay(P,Q) - \\ \quad II * Distance(P,Q)), \ otherwise \end{cases}$$

- Note that only scheduled predecessors will be considered in the computation of *Estart*

Rotating Register Set and Modulo-Variable Expansion

- 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 i1 in time step 2 is used by i2 only in time step 5
    - However, another instance of i1 from *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

Time Step	Iter. 0	Iter. 1	Iter. 2	
0	i0 : ld			
1				Prolog
2	i1 : mult	i0 : ld		
3	i3 : add			
4	i4 : sub	i1 : mult	i0 : ld	Kernel
5	i2 : st i5 : bge	i3 : add		
6		i4 : sub	i1 : mult	
7		i2 : st i5 : bge	i3 : add	Epilog
8			i4 : sub	
9			i2 : st i5 : bge	

A Software Pipelined Schedule with II = 2

Register Spilling in Software Pipelining

- 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 them 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)