15-745

Software Pipelining

(some slides borrowed from T Callahan & M. Voss)

Software Pipelining

- Software pipelining is an IS technique that reorders the instructions in a loop.
  - Possibly moving instructions from one iteration to the previous or the next iteration.
  - Very large improvements in running time are possible.
- The first serious approach to software pipelining was presented by Aiken & Nicolau.
  - Aiken's 1988 Ph.D. thesis.
  - Impractical as it ignores resource hazards (focusing only on data-dependence constraints).
    - But sparked a large amount of follow-on research.

Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration

Assume all have latency of 2

```
A: a ←  ld [d]
B: b ←  a * a
C:      st [d], b
D: d ←  d + 4
```

A		B		C		D	

Can we decrease the latency?

- Lets unroll

```
A:  a ← ld [d]
B:  b ← a * a
C:      st [d], b
D:  d ← d + 4
A1: a ← ld [d]
B1: b ← a * a
C1:     st [d], b
D1: d ← d + 4
```

A		B		C		D		A1		B1		C1		D1	

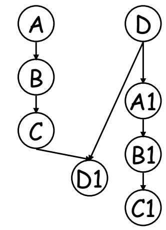
Rename variables

```
A:  a ←   ld [d]
B:  b ←   a * a
C:        st [d], b
D:  d1 ←  d + 4
A1: a1 ←  ld [d1]
B1: b1 ←  a1 * a1
C1:       st [d1], b1
D1: d ←   d1 + 4
```

A		B		C		D		A1		B1		C1		D1	

Schedule

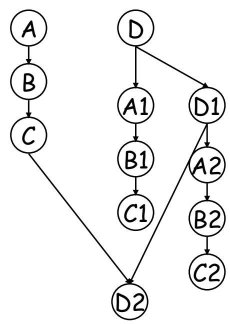
```
A:  a ←   ld [d]
B:  b ←   a * a
C:        st [d], b
D:  d1 ←  d + 4
A1: a1 ←  ld [d1]
B1: b1 ←  a1 * a1
C1:       st [d1], b1
D1: d ←   d1 + 4
```



A		B		C		D1	
D		A1		B1		C1	

Unroll Some More

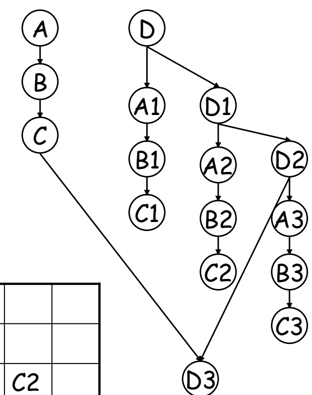
```
A:  a ←   ld [d]
B:  b ←   a * a
C:        st [d], b
D:  d1 ←  d + 4
A1: a1 ←  ld [d1]
B1: b1 ←  a1 * a1
C1:       st [d1], b1
D1: d2 ←  d1 + 4
A2: a2 ←  ld [d2]
B2: b2 ←  a2 * a2
C2:       st [d2], b2
D2: d ←   d2 + 4
```



A		B		C		D2	
D		A1		B1		C1	
	D1		A2		B2		C2

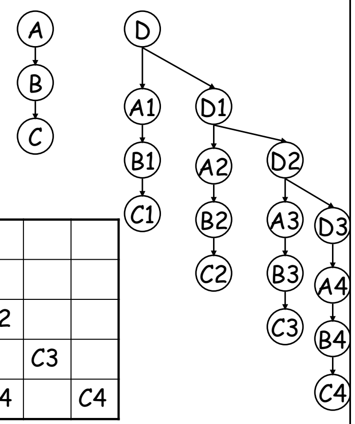
Unroll Some More

```
A:  a ←   ld [d]
B:  b ←   a * a
C:        st [d], b
D:  d1 ←  d + 4
A1: a1 ←  ld [d1]
B1: b1 ←  a1 * a1
C1:       st [d1], b1
D1: d2 ←  d1 + 4
A2: a2 ←  ld [d2]
B2: b2 ←  a2 * a2
C2:       st [d2], b2
D2: d ←   d2 + 4
```



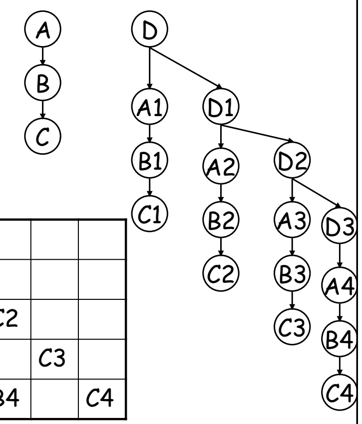
A		B		C		D3		
D		A1		B1		C1		
	D1		A2		B2		C2	
		D2		A3		B3		C3

One More Time



A		B		C		D4			
D		A1		B1		C1			
	D1		A2		B2		C2		
		D2		A3		B3		C3	
			D3		A4		B4		C4

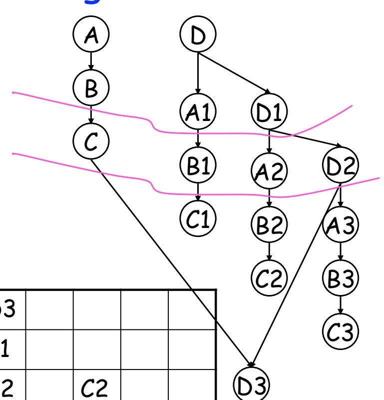
Can Rearrange



A		B		C		D4			
D		A1		B1		C1			
	D1		A2		B2		C2		
		D2		A3		B3		C3	
			D3		A4		B4		C4

Rearrange

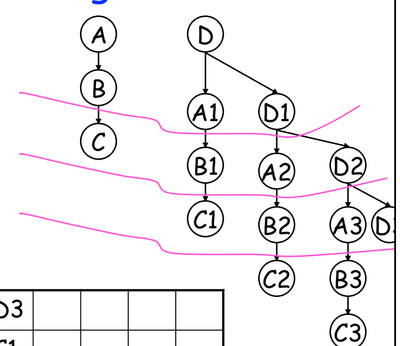
```
A:   a ←   ld [d]
B:   b ←   a * a
C:         st [d], b
D:   d1 ←  d + 4
A1:  a1 ←  ld [d1]
B1:  b1 ←  a1 * a1
C1:        st [d1], b1
D1:  d2 ←  d1 + 4
A2:  a2 ←  ld [d2]
B2:  b2 ←  a2 * a2
C2:        st [d2], b2
D2:  d ←   d2 + 4
```



A		B		C		D3				
D		A1		B1		C1				
		D1		A2		B2		C2		
				D2		A3		B3		C3

Rearrange

```
A:   a ←   ld [d]
B:   b ←   a * a
C:         st [d], b
D:   d1 ←  d + 4
A1:  a1 ←  ld [d1]
B1:  b1 ←  a1 * a1
C1:        st [d1], b1
D1:  d2 ←  d1 + 4
A2:  a2 ←  ld [d2]
B2:  b2 ←  a2 * a2
C2:        st [d2], b2
D2:  d ←   d2 + 4
```



A		B		C		D3				
D		A1		B1		C1				
		D1		A2		B2		C2		
				D2		A3		B3		C3

SP Loop

```
A:   a ←    ld [d]
B:   b ←    a * a
D:   d1 ←   d + 4
A1:  a1 ←   ld [d1]
D1:  d2 ←   d1 + 4

C:          st [d], b
B1:  b1 ←   a1 * a1
A2:  a2 ←   ld [d2]
D2:  d ←    d2 + 4

B2:  b2 ←   a2 * a2
C1:         st [d1], b1
D3:  d2 ←   d1 + 4
C2:         st [d2], b2
```

Prolog

Body

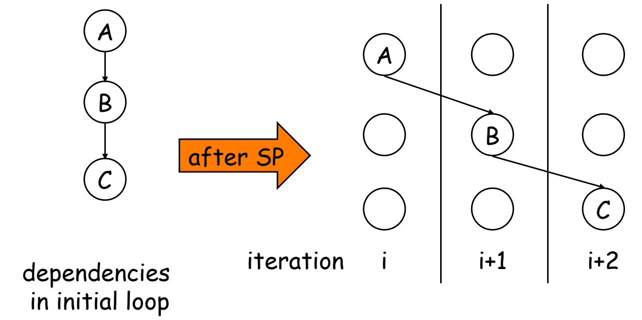
Epilog

A		B		C	C	C	D3		
D		A1		B1	B1	B1	C1		
		D1		A2	A2	A2	B2		C2
				D2	D2	D2			

 13

Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration



 14

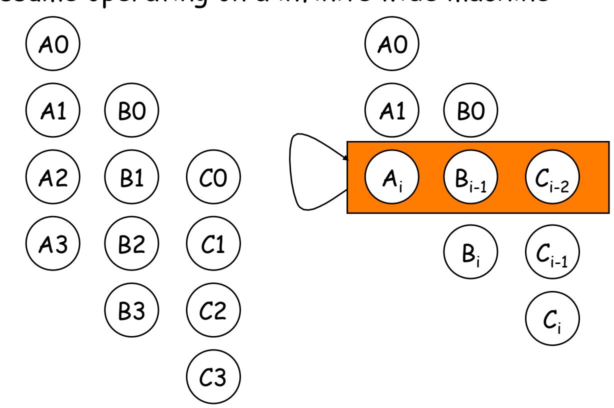
Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration
- But also, to uncover ILP across iteration boundaries!

 15

Example

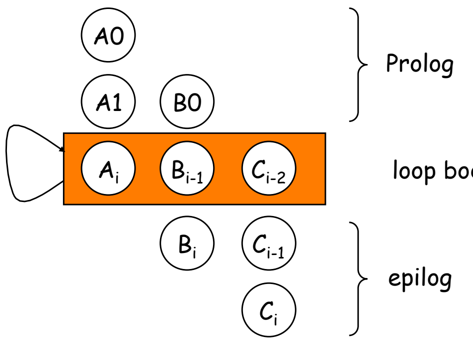
Assume operating on a infinite wide machine



 16

Example

Assume operating on a infinite wide machine



Dealing with exit conditions

```
for (i=0; i<N; i++)
{
    Ai
    Bi
    Ci
}
```

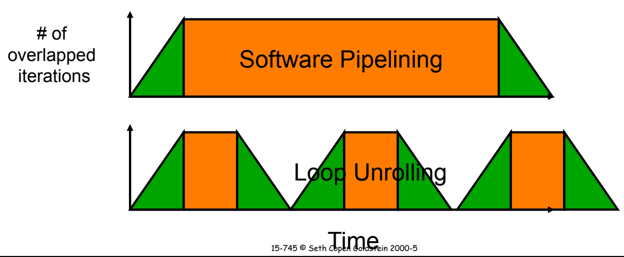
```
i=0
if (i >= N) goto done
A0
B0
if (i+1 == N) goto last
i=1
A1
if (i+2 == N) goto epilog
i=2
```

```
loop:
    Ai
    Bi-1
    Ci-2
    i++
    if (i < N) goto loop
epilog:
    Bi
    Ci-1
last:
    ci
done:
```


Loop Unrolling V. SP

For SuperScalar or VLIW

- Loop Unrolling reduces loop overhead
- Software Pipelining reduces fill/drain
- Best is if you combine them



Aiken/Nicolau Scheduling
Step 1

Perform scalar replacement to eliminate memory references where possible.

```
for i:=1 to N do
    a := j ⊕ V[i-1]
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

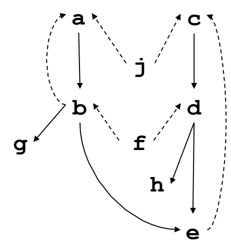
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

Aiken/Nicolau Scheduling
Step 2

Unroll the loop and compute the data-dependence graph (DDG).

DDG for rolled loop:

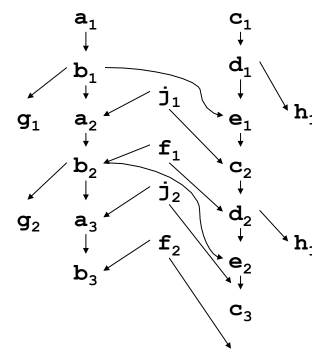
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```



Aiken/Nicolau Scheduling
Step 2, cont'd

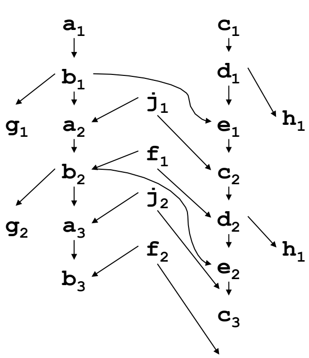
DDG for unrolled loop:

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```



Aiken/Nicolau Scheduling
Step 3

Build a tableau of iteration number vs cycle time.

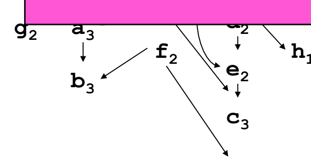


cycle	iteration 1	2	3	4	5	6
1	acfj	fj	fj	fj	fj	fj
2	bd					
3	egh	a				
4		cb				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	a
12				eh		b
13					c	g
14					d	
15					eh	

Aiken/Nicolau Scheduling
Step 3

basically, you're emulating a superscalar with infinite resources, infinite register renaming, always predicting the loop-back branch: thus, just pure data dependency

...on number vs cycle time.



cycle	iteration 1	2	3	4	5	6
1	acfj	fj	fj	fj	fj	fj
2	bd					
3	egh	a				
4		cb				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	a
12				eh		b
13					c	g
14					d	
15					eh	

Aiken/Nicolau Scheduling Step 4

Find repeating patterns of instructions.

cycle \ iteration	1	2	3	4	5	6
1	acfj	fj	fj	fj	fj	fj
2	bd					
3	egh	a				
4		cb				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	a
12				eh		b
13					c	g
14					d	
15					eh	

Aiken/Nicolau Scheduling Step 4

Find repeating patterns of instructions.

cycle \ iteration	1	2	3	4	5	6
1	acfj	fj	fj	fj	fj	fj
2	bd					
3	egh	a				
4		cb				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	a
12				eh		b
13					c	g
14					d	
15					eh	

Aiken/Nicolau Scheduling Step 4

Find repeating patterns of instructions.

cycle \ iteration	1	2	3	4	5	6
1	acfj	fj	fj	fj	fj	fj
2	bd					
3	egh	a				
4		cb				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	a
12				eh		b
13					c	g
14					d	
15					eh	

Go back and relate slopes to DDG

Aiken/Nicolau Scheduling Step 5

"Coalesce" the slopes.

cycle \ iteration	1	2	3	4	5	6
1	acfj	fj	fj	fj	fj	fj
2	bd					
3	egh	a				
4		cb				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	a
12				eh		b
13					c	g
14					d	
15					eh	

cycle \ iteration	1	2	3	4	5	6
1	acfj					
2	bd	fj				
3	egh	a				
4		cb	fj			
5		dg	a			
6		eh	b	fj		
7			cg	a		
8			d	b		
9			eh	g	fj	
10				c	a	
11				d	b	
12				eh	g	
13					c	
14					d	
15					eh	

Aiken/Nicolau Scheduling Step 6

Find the loop body and "reroll" the loop.

cycle	iteration 1	2	3	4	5	6
1	acfj					
2	bd	fj				
3	egh	a				
4		cb	fj			
5		dg	a			
6		eh	b	fj		
7			cg	a		
8			d	b		
9			eh	g	fj	
10				c	a	
11				d	b	
12				eh	g	
13					c	
14					d	
15					eh	

Aiken/Nicolau Scheduling Step 6

Find the loop body and "reroll" the loop.

cycle	iteration 1	2	3	4	5	6	
1	acfj						
2	bd	fj					
3	egh	a					← Prologue/entry code
4		cb	fj				
5		dg	a				
6		eh	b	fj			
7			cg	a			
8			d	b			
9			eh	g	fj		← Loop body (boxed: cycles 8–10)
10				c	a		
11				d	b		
12				eh	g		
13					c		← Epilogue/exit code
14					d		
15					eh		

Aiken/Nicolau Scheduling Step 7

Generate code.
(Assume VLIW-like machine for this example. The instructions on each line should be issued in parallel.)

```
   a1 := j0 ⊕ b0     c1 := e0 ⊕ j0     f1 := U[1]      j1 := X[1]
   b1 := a1 ⊕ f0     d1 := f0 ⊕ c1     f2 := U[2]      j2 := X[2]
   e1 := b1 ⊕ d1     V[1] := b1        W[1] := d1      a2 := j1 ⊕ b1
   c2 := e1 ⊕ j1     b2 := a2 ⊕ f1     f3 := U[3]      j3 := X[3]
   d2 := f1 ⊕ c2     V[2] := b2        a3 := j2 ⊕ b2
   e2 := b2 ⊕ d2     W[2] := d2        b3 := a3 ⊕ f2   f4 := U[4]    j4 := X[4]
   c3 := e2 ⊕ j2     V[3] := b3        a4 := j3 ⊕ b3    i := 3
L:
   d_i := f_i-1 ⊕ c_i     b_i+1 := a_i ⊕ f_i
   e_i := b_i ⊕ d_i       W[i] := d_i            V[i+1] := b_i+1   f_i+2 := U[I+2]  j_i+2 := X[i+2]
   c_i+1 := e_i ⊕ j_i     a_i+2 := j_i+1 ⊕ b_i+1 i := i+1          if i<N-2 goto L

   d_N-1 := f_N-2 ⊕ c_N-1 b_N := a_N ⊕ f_N-1
   e_N-1 := b_N-1 ⊕ d_N-1 W[N-1] := d_N-1   v[N] := b_N
   c_N := e_N-1 ⊕ j_N-1
   d_N := f_N-1 + c_N
   e_N := b_N ⊕ d_N       w[N] := d_N
```


Aiken/Nicolau Scheduling Step 8

- Since several versions of a variable (e.g., j_i and j_{i+1}) might be live simultaneously, we need to add new temps and moves

```
   a1 := j0 ⊕ b0     c1 := e0 ⊕ j0     f1 := U[1]      j1 := X[1]
   b1 := a1 ⊕ f0     d1 := f0 ⊕ c1     f2 := U[2]      j2 := X[2]
   e1 := b1 ⊕ d1     V[1] := b1        W[1] := d1      a2 := j1 ⊕ b1
   c2 := e1 ⊕ j1     b2 := a2 ⊕ f1     f3 := U[3]      j3 := X[3]
   d2 := f1 ⊕ c2     V[2] := b2        a3 := j2 ⊕ b2
   e2 := b2 ⊕ d2     W[2] := d2        b3 := a3 ⊕ f2   f4 := U[4]    j4 := X[4]
   c3 := e2 ⊕ j2     V[3] := b3        a4 := j3 ⊕ b3    i := 3
L:
   d_i := f_i-1 ⊕ c_i     b_i+1 := a_i ⊕ f_i
   e_i := b_i ⊕ d_i       W[i] := d_i            V[i+1] := b_i+1   f_i+2 := U[I+2]  j_i+2 := X[i+2]
   c_i+1 := e_i ⊕ j_i     a_i+2 := j_i+1 ⊕ b_i+1 i := i+1          if i<N-2 goto L

   d_N-1 := f_N-2 ⊕ c_N-1 b_N := a_N ⊕ f_N-1
   e_N-1 := b_N-1 ⊕ d_N-1 W[N-1] := d_N-1   v[N] := b_N
   c_N := e_N-1 ⊕ j_N-1
   d_N := f_N-1 + c_N
   e_N := b_N ⊕ d_N       w[N] := d_N
```

Aiken/Nicolau Scheduling Step 8

- Since several versions of a variable (e.g., j_i and j_{i+1}) might be live simultaneously, we need to add new temps and moves

```
    a1  := j0 ⊕ b0    c1  := e0 ⊕ j0    f1  := U[1]      j1  := X[1]
    b1  := a1 ⊕ f0    d1  := f0 ⊕ c1    f''  := U[2]      j2  := X[2]
    e1  := b1 ⊕ d1    V[1]  := b1       W[1]  := d1      a2  := j1 ⊕ b1
    c2  := e1 ⊕ j1    b2  := a2 ⊕ f1    f'  := U[3]      j'  := X[3]
    d2  := f1 ⊕ c2    V[2]  := b2       a3  := j2 ⊕ b2
    e2  := b2 ⊕ d2    W[2]  := d2       b3  := a3 ⊕ f''   f4  := U[4]    j4 := X[4]
    c3  := e2 ⊕ j2    V[3]  := b3       a4  := j' ⊕ b3    i  := 3
L:
    d_i  := f'' ⊕ c_i      b_i+1  := a' ⊕ f'    b'  := b; a'=a; f''=f'; f'=f; j''=j'; j'=j
    e_i  := b' ⊕ d_i       W[i]  := d_i         V[i+1]  := b_i+1   f_i+2  := U[I+2]   j_i+2  := X[i+2]
    c_i+1  := e_i ⊕ j'     a_i+2  := j'' ⊕ b_i+1  i  := i+1        if i<N-2 goto L

    d_N-1  := f_N-2 ⊕ c_N-1  b_N  := a_N ⊕ f_N-1
    e_N-1  := b_N-1 ⊕ d_N-1  W[N-1]  := d_N-1    v[N]  := b_N
    c_N  := e_N-1 ⊕ j_N-1
    d_N  := f_N-1 + c_N
    e_N  := b_N ⊕ d_N        w[N]  := d_N
```

15-745 © Seth Copen Goldstein 2000-5 33

Next Step in SP

- AN88 did not deal with resource constraints.
- Modulo Scheduling is a SP algorithm that does.
- It schedules the loop based on
  - resource constraints
  - precedence constraints
- Basically, it's list scheduling that takes into account resource conflicts from overlapping iterations

15-745 © Seth Copen Goldstein 2000-5 34

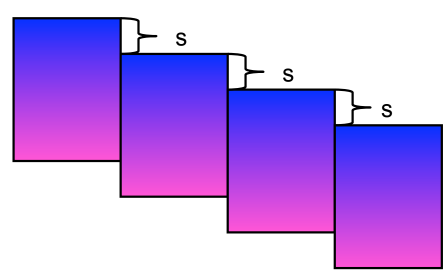
Resource Constraints

- Minimally indivisible sequences, **i** and **j**, can execute together if combined resources in a step do not exceed available resources.
- R(i) is a resource configuration vector
  R(i) is the number of units of resource i
- r(i) is a resource usage vector s.t.
  $0 \le r(i) \le R(i)$
- Each node in G has an associated r(i)

15-745 © Seth Copen Goldstein 2000-5 35

Software Pipelining Goal

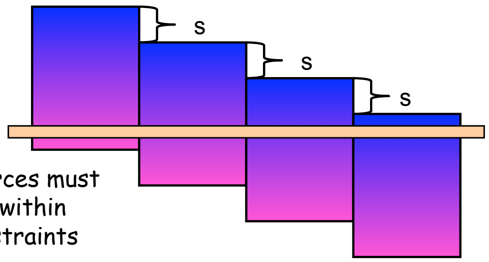
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, **s**
- Goal: minimize **s**.



15-745 © Seth Copen Goldstein 2000-5 36

Software Pipelining Goal

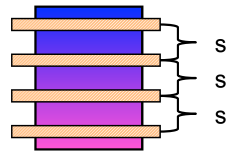
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, **s**
- Goal: minimize **s**.



resources must be within constraints

Software Pipelining Goal

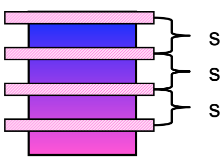
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, **s**
- Goal: minimize **s**.



resources must be within constraints

Software Pipelining Goal

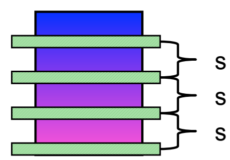
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, **s**
- Goal: minimize **s**.



resources must be within constraints

Software Pipelining Goal

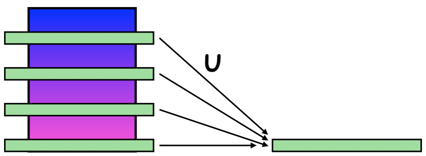
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, **s**
- Goal: minimize **s**.



resources must be within constraints

Software Pipelining Goal

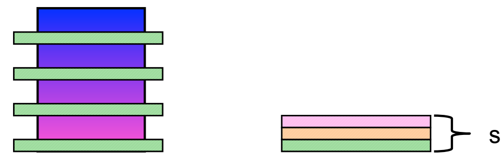
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, **s**
- Goal: minimize **s**.



resources must be within constraints

Software Pipelining Goal

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, **s**
- Goal: minimize **s**.



resources must be within constraints

modulo resource table

Precedence Constraints

- Review: for acyclic scheduling, constraint is just the required delay between two ops u, v: <d(u,v)>
- For an edge, $u \rightarrow v$, we must have
  $\sigma(v)-\sigma(u) \geq d(u,v)$

Precedence Constraints

- Cyclic: constraint becomes a tuple: <p,d>
  - p is the minimum iteration delay (or the loop carried dependence distance)
  - d is the delay
- For an edge, $u \rightarrow v$, we must have
  $\sigma(v)-\sigma(u) \geq d(u,v)-s*p(u,v)$
- $p \geq 0$
- If data dependence is
  - within an iteration, p=0
  - loop-carried across p iter boundaries, p>0

Iterative Approach

- Finding minimum S that satisfies the constraints is NP-Complete.
- Heuristic:
  - Find lower and upper bounds for S
  - foreach s from lower to upper bound?
    - Schedule graph.
    - If succeed, done
    - Otherwise try again (with next higher s)
- Thus: "Iterative Modulo Scheduling" Rau, et.al.

Iterative Approach

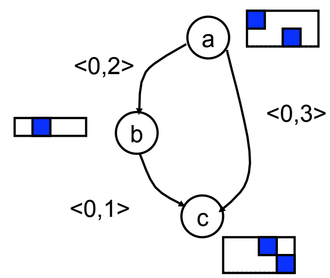
- Heuristic:
  - Find lower and upper bounds for S
  - foreach s from lower to upper bound
    - Schedule graph.
    - If succeed, done
    - Otherwise try again (with next higher s)
- So the key difference:
  - AN88 does not assume S when scheduling
  - IMS must assume an S for each scheduling attempt to understand resource conflicts

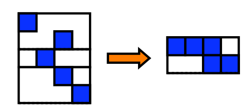
Lower Bounds

- Resource Constraints: S_R (also called II_{res}) maximum over all resources of # of uses divided by # available... rounded up or down?
- Precedence Constraints: S_E (also called II_{rec}) max over all cycles: d(c)/p(c)

In practice, one is easy, other is hard.
Tim's secret approach: just use S_R as lower bound, then do binary search for best S

Acyclic Example



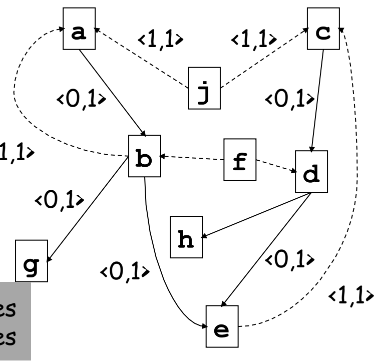


Lower Bound: S_R=2
Upper Bound: 5

Lower Bound on s

- Assume 1 ALU and 1 MU
- Assume latency Op or load is 1 cycle

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```



Resources => 5 cycles
Dependencies => 3 cycles

Scheduling data structures

To schedule for initiation interval s:

- Create a resource table with s rows and R columns
- Create a vector, σ, of length N for n instructions in the loop
  - σ[n] = the time at which n is scheduled, or NONE
- Prioritize instructions by some heuristic
  - critical path (or cycle)
  - resource critical

Scheduling algorithm

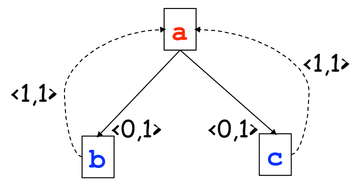
- Pick an instruction, n
- Calculate earliest time due to dependence constraints
  For all x=pred(n),
  earliest = max(earliest, σ(x)+d(x,n)-s·p(x,n))
- try and schedule n from earliest to (earliest+s-1) s.t. resource constraints are obeyed.
  - possible twist: deschedule a conflicting node to make way for n, maybe randomly, like sim anneal
- If we fail, then this schedule is faulty (i.e. give up on this s)

Scheduling algorithm – cont.

- We now schedule n at earliest, I.e., σ(n) = earliest
- Fix up schedule
  - Successors, x, of n must be scheduled s.t. σ(x) >= σ(n)+d(n,x)-s·p(n,x), otherwise they are removed (descheduled) and put back on worklist.
- repeat this some number of times until either
  - succeed, then register allocate
  - fail, then increase s

Simplest Example

```
for () {
  a = b+c
  b = a*a
  c = a*194
}
```

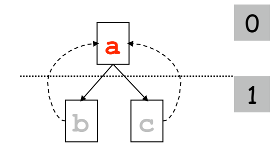


Resources: 1 1

What is IIres?
What is IIrec?

Simplest Example

```
for () {
  a = b+c
  b = a*a
  c = a*194
}
```



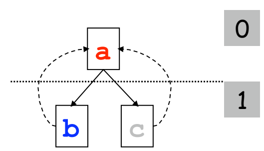
Try II = 2

Modulo Resource Table:

0	1	
1		

Simplest Example

```
for () {
  a = b+c
  b = a*a
  c = a*194
}
```



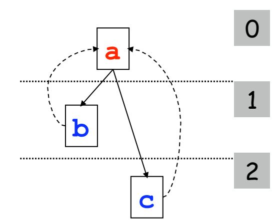
Try II = 2

Modulo Resource Table:

0	1	
1		1

Simplest Example

```
for () {
  a = b+c
  b = a*a
  c = a*194
}
```



Try II = 2

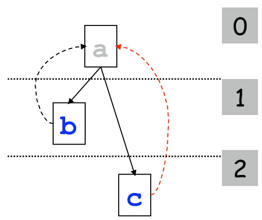
Modulo Resource Table:

0	1	1
1		1

Simplest Example

```
for () {
  a = b+c
  b = a*a
  c = a*194
}
```

Try II = 2



Modulo Resource Table:

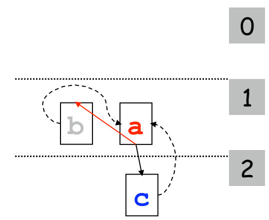
0		1
1		1

earliest a: sigma(c) + delay(c) - 2
= 2+1-2 = 1

Simplest Example

```
for () {
  a = b+c
  b = a*a
  c = a*194
}
```

Try II = 2



Modulo Resource Table:

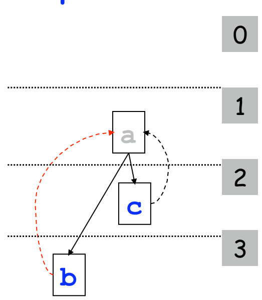
0		1
1	1	

earliest b?
scheduled b?
what next?

Simplest Example

```
for () {
  a = b+c
  b = a*a
  c = a*194
}
```

Try II = 2



Modulo Resource Table:

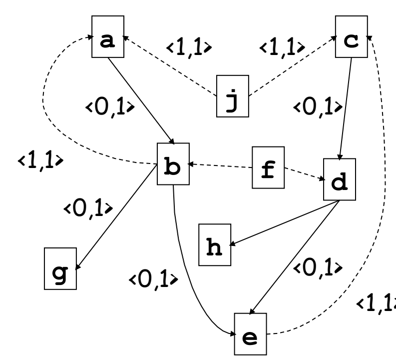
0		1
1		1

Lesson: lower bound
may not be achievable

Example

```
for i:=1 to N do
     a := j ⊕ b
     b := a ⊕ f
     c := e ⊕ j
     d := f ⊕ c
     e := b ⊕ d
     f := U[i]
  g: V[i]  := b
  h: W[i]  := d
     j := X[i]
```

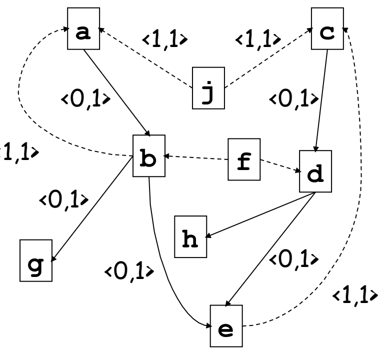
Priorities: ?



Example

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

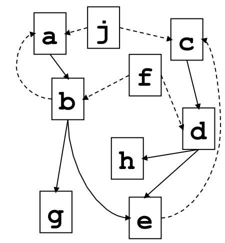
Priorities: c,d,e,a,b,f,j,g,h



```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

s=5

Priorities: c,d,e,a,b,f,j,g,h



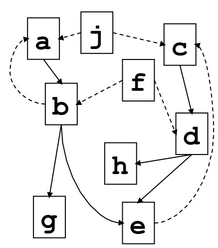
ALU	MU

instr	σ
a	
b	
c	
d	
e	
f	
g	
h	
j	

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

s=5

Priorities: a,b,f,j,g,h



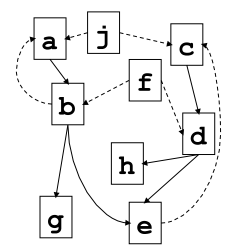
ALU	MU
c	
d	
e	

instr	σ
a	
b	
c	0
d	1
e	2
f	
g	
h	
j	

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

s=5

Priorities: b,f,j,g,h



ALU	MU
c	
d	
e	
a	

instr	σ
a	3
b	
c	0
d	1
e	2
f	
g	
h	
j	

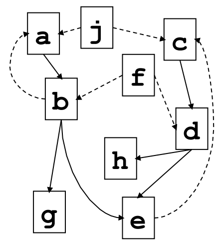
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

s=5

ALU	MU
c	
d	
e	
a	
b	

instr	σ
a	3
b	4
c	0
d	1
e	2
f	
g	
h	
j	

Priorities: b,f,j,g,h



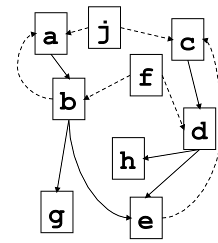

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

s=5

ALU	MU
c	
d	
a	
b	

instr	σ
a	3
b	4
c	0
d	1
e	
f	
g	
h	
j	

Priorities: e,f,j,g,h



b causes b->e edge violation

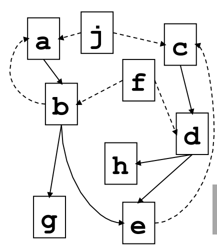

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

s=5

ALU	MU
c	
d	
e	
a	
b	

instr	σ
a	3
b	4
c	0
d	1
e	7
f	
g	
h	
j	

Priorities: e,f,j,g,h



e causes e->c edge violation

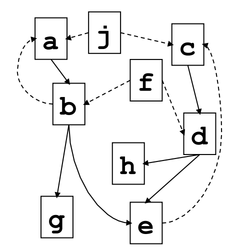

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

s=5

ALU	MU
c	f
d	
e	
a	
b	

instr	σ
a	3
b	4
c	5
d	6
e	7
f	0
g	
h	
j	

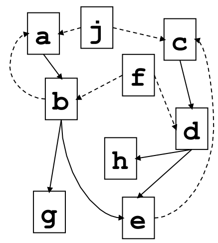
Priorities: f,j,g,h




```
for i:=1 to N do
     a := j ⊕ b
     b := a ⊕ f
     c := e ⊕ j
     d := f ⊕ c
     e := b ⊕ d
     f := U[i]
 g: V[i] := b
 h: W[i] := d
     j := X[i]
```

s=5

Priorities:j,g,h



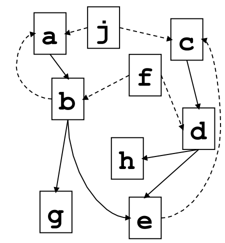
ALU	MU
c	f
d	j
e	
a	
b	

instr	σ
a	3
b	4
c	5
d	6
e	7
f	0
g	
h	
j	1


```
for i:=1 to N do
     a := j ⊕ b
     b := a ⊕ f
     c := e ⊕ j
     d := f ⊕ c
     e := b ⊕ d
     f := U[i]
 g: V[i] := b
 h: W[i] := d
     j := X[i]
```

s=5

Priorities:g,h



ALU	MU
c	f
d	j
e	g
a	h
b	

instr	σ
a	3
b	4
c	5
d	6
e	7
f	0
g	7
h	8
j	1

Creating the Loop

- Create the body from the schedule.
- Determine which iteration an instruction falls into
  - Mark its sources and dest as belonging to that iteration.
  - Add Moves to update registers
- Prolog fills in gaps at beginning
  - For each move we will have an instruction in prolog, and we fill in dependent instructions
- Epilog fills in gaps at end

instr	σ
a	3
b	4
c	5
d	6
e	7
f	0
g	7
h	8
j	1


```
f0 = U[0];
j0 = X[0];

FOR i = 0 to N
   f1 := U[i+1]
   j1 := X[i+1]
   nop
   a := j0 ? b
   b := a ? f0
   c := e ? j0
   d := f0 ? c
   e := b ? d            g: V[i] := b
h: W[i] := d
   f0 = f1
   j0 = j1
```

Conditionals

- What about internal control structure, I.e., conditionals
- Three approaches
  - Schedule both sides and use conditional moves
  - Schedule each side, then make the body of the conditional a macro op with appropriate resource vector
  - Trace schedule the loop

What to take away

- Dependence analysis is very important
- Software pipelining is cool
- Registers are a key resource