Instruction Scheduling
Software Pipelining

15-411/15-611 Compiler Design

Seth Copen Goldstein

November 19, 2020

Instruction-level Parallelism

- Most modern processors have the ability to execute several adjacent instructions simultaneously.
  - Pipelined machines.
  - Very-long-instruction-word machines (VLIW).
  - Superscalar machines.
  - Dynamic scheduling/out-of-order machines.
- ILP is limited by several kinds of *execution constraints*:
  - Data dependence constraints.
  - Resource constraints ("hazards")

Execution Constraints

- Data-dependence constraints:
  - If instruction A computes a value that is read by instruction B, then B cannot execute before A is completed.
- Resource hazards:
  - Limited # of functional units.
    - If there are *n* functional un multipliers), then only *n* ins of unit can execute at once
  - Limited instruction issue.
    - If the instruction-issue unit can issue only *n* instructions at a time, then this limits ILP.
  - Limited register set.
    - Any schedule of instructions must have a valid register allocation.

For example:

```
ld      %rsp(-28), %rdi

add     %rdi, %rax
```

Instruction Scheduling

- The purpose of instruction scheduling (IS) is to order the instructions for maximum ILP.
  - Keep all resources busy every cycle.
  - If necessary, eliminate data dependences and resource hazards to accomplish this.
- The IS problem is NP-complete (and bad in practice).
  - So heuristic methods are necessary.

Instruction Scheduling

- There are *many* different techniques for IS.
  - Still an open area of research.
- Most optimizing compilers perform good local IS, and only simple global IS.
- The biggest opportunities are in scheduling the code for loops.
  - "Software pipelining" is an attractive idea, though not yet widely used in practical compilers.

Should the Compiler Do IS?

- Many modern machines perform dynamic reordering of instructions.
  - Also called “out-of-order execution” (OOOE).
  - Not yet clear whether this is a good idea.
  - Pro:
    - OOOE can use additional registers and register renaming to eliminate data dependences that no amount of static IS can accomplish.
    - No need to recompile programs when hardware changes.
  - Con:
    - OOOE means more complex hardware (and thus longer cycle times and more wattage).
    - And can’t be optimal since IS is NP-complete.

What we will cover

- Scheduling basic blocks
  - List scheduling
  - Long-latency operations
  - Delay slots
- Software Pipelining

- What we need to know
  - data dependencies
  - register renaming
  - scalar replacement

Defining Dependencies

• Flow Dependence	W → R	δ^f	} true	
• Anti-Dependence	R → W	δ^a	} false	
• Output Dependence	W → W	δ^o		
• Input Dependence	R → R	δ^i		

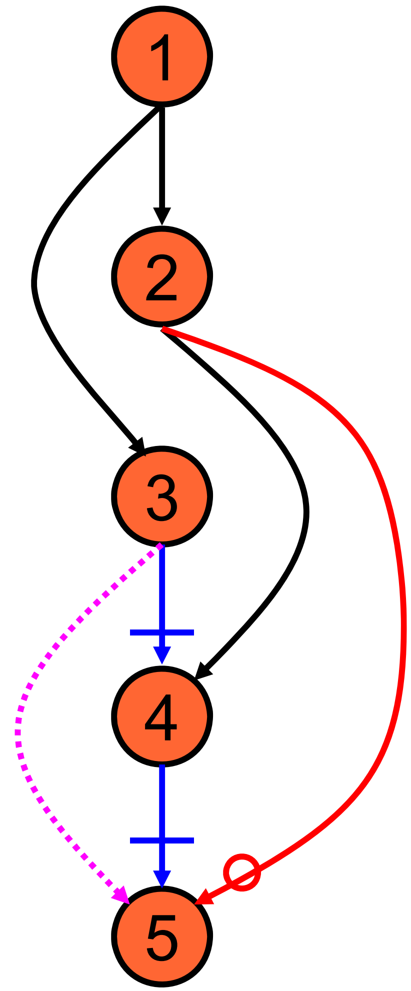
```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

Not generally defined

Example Dependencies

```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

S1 δ^f S2	due to a
S1 δ^f S3	due to a
S2 δ^f S4	due to b
S3 δ^a S4	due to d
S4 δ^a S5	due to b
S2 δ^o S5	due to b
S3 δ^i S5	due to a



Renaming of Variables

- Sometimes constraints are not “real,” in the sense that a simple renaming of variables/registers can eliminate them.
  - Output dependence (WW):
    A and B write to the same variable.
  - Anti-dependence (RW):
    A reads from a variable to which B writes.
- In such cases, the order of A and B cannot be changed unless variables are renamed.
  - Can sometimes be done by the hardware, to a limited extent.

Register Renaming Example

r1 ← r2 + 1	r7 ← r2 + 1	r7 ← r2 + 1
[fp+8] ← r1	[fp+8] ← r7	r1 ← r3 + 2
r1 ← r3 + 2	r1 ← r3 + 2	[fp+8] ← r7
[fp+12] ← r1	[fp+12] ← r1	[fp+12] ← r1

- Can perform register renaming after register allocation
  - Constrained by available registers
  - Constrained by live on entry/exit
- Instead, do scheduling before register allocation

Scheduling a BB

$w \leftarrow w * 2 * x * y * z$

r1	←	[fp+w]
r2	←	2
r1	←	r1 * r2
r2	←	[fp+x]
r1	←	r1 * r2
r2	←	[fp+y]
r1	←	r1 * r2
r2	←	[fp+z]
r1	←	r1 * r2
[fp+w]	←	r1

- What do we need to know?
  - Latency of operations
  - # of registers
- Assume:
  - load 5
  - store 5
  - mult 2
  - others 1
- Also assume,
  - operations are non-blocking

Scheduling a BB

Assume:

- load 5
- store 5
- mult 2
- others 1
- operations are non-blocking

w ← w * 2 * x * y * z

1	r1	← [fp+w]		
2	r2	← 2		
6	r1	← r1 * r2		*w*2*
7	r2	← [fp+x]		
12	r1	← r1 * r2		*w*2*x*
13	r2	← [fp+y]		
18	r1	← r1 * r2		*w*2*x*y*
19	r2	← [fp+z]		*w*2*x*y*z*
24	r1	← r1 * r2		
26	[fp+w]	← r1		
33	r1 can be used again			

We can do better

- Assume:
  - load 5
  - store 5
  - mult 2
  - others 1
  - operations are non-blocking

1	r1	← [fp+w]		
2	r2	← [fp+x]		
3	r3	← [fp+y]		
4	r4	← [fp+z]		
5	r5	← 2		
6	r1	← r1 * r5		*w*2*
8	r1	← r1 * r2		*w*2*x*
10	r1	← r1 * r3		*w*2*x*y*
12	r1	← r1 * r4		*w*2*x*y*z*
14	[fp+w]	← r1		
19	r1 can be used again			

We can do even better if we assume what?

Defining Better

1	r1	← [fp+w]
2	r2	← 2
6	r1	← r1 * r2
7	r2	← [fp+x]
12	r1	← r1 * r2
13	r2	← [fp+y]
18	r1	← r1 * r2
19	r2	← [fp+z]
24	r1	← r1 * r2
26	[fp+w]	← r1
33	r1 can be used again	

1	r1	← [fp+w]
2	r2	← [fp+x]
3	r3	← [fp+y]
4	r4	← [fp+z]
5	r5	← 2
6	r1	← r1 * r5
8	r1	← r1 * r2
10	r1	← r1 * r3
12	r1	← r1 * r4
14	[fp+w]	← r1
19	r1 can be used again	

The Scheduler

- Given:
  - Code to schedule
  - Resources available (FU and # of Reg)
  - Latencies of instructions
- Goal:
  - Correct code
  - Better code [fewer cycles, less power, fewer registers, …]
  - Do it quickly

More Abstractly

- Given a graph G = (V,E) where
  - nodes are operations
    - Each operation has an associated delay and type
  - edges between nodes represent dependencies
  - The number of resources of type t, R(t)
- A schedule assigns to each node a cycle number:
  - $S(n) \geq 0$
  - If $(n,m) \in G$, $S(m) \geq S(n) + delay(n)$
  - $|\{ n \mid S(n) = x \text{ and } type(n) = t\}| <= R(t)$
- Goal is shortest length schedule, where length
  - L(S) = max over n, S(n)+delay(n)

List Scheduling

- Keep a list of available instructions, I.e.,
  - If we are at cycle k, then all predecessors, p, in graph have all been scheduled so that $S(p)+delay(p) \leq k$
- Pick some instruction, n, from queue such that there are resources for type(n)
- Update available instructions and continue

- It is all in how we pick instructions

Lots of Heuristics

- forward or backward
- choose instructions on critical path
- ASAP or ALAP
- Balanced paths
- depth in schedule graph

Delayed Load Scheduling

- Aim: avoid pipeline hazards in load/store unit
  - load followed by use of target reg
  - store followed by load
- Simplifies in two ways
  - 1 cycle latency for load/store
  - includes all dependencies (WaW included)

The algorithm

- Construct Scheduling dag
- Make srcs of dag candidates
- Pick a candidate
  - Choose an instruction with an interlock
  - Choose an instruction with a large number of successors
  - Choose with longest path to root
- Add newly available instruction to candidate list

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.
  - 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	

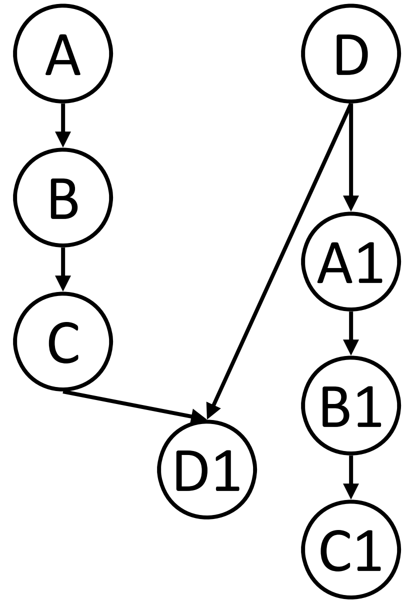
Rename variables

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

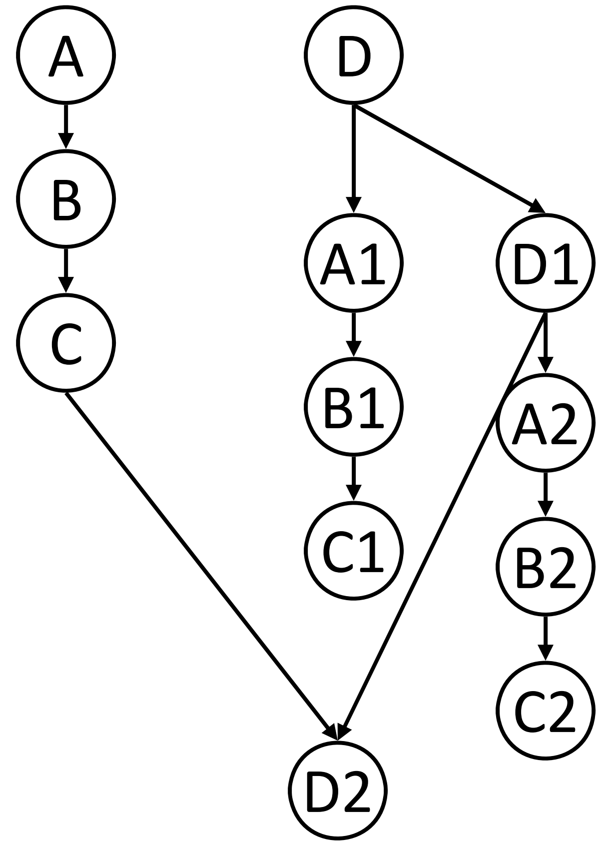
```
A:   a  ←   ld [d]
B:   b  ←   a * a
C:          st [d], b
D:   d1 ←   d + 4
A1:  a1 ←   ld [d1]
B1:  b1 ←   a1 * a1
C1:         st [d1], b1
D1:  d  ←   d1 + 4
```



A		B		C		D1	
D		A1		B1		C1	

Unroll Some More

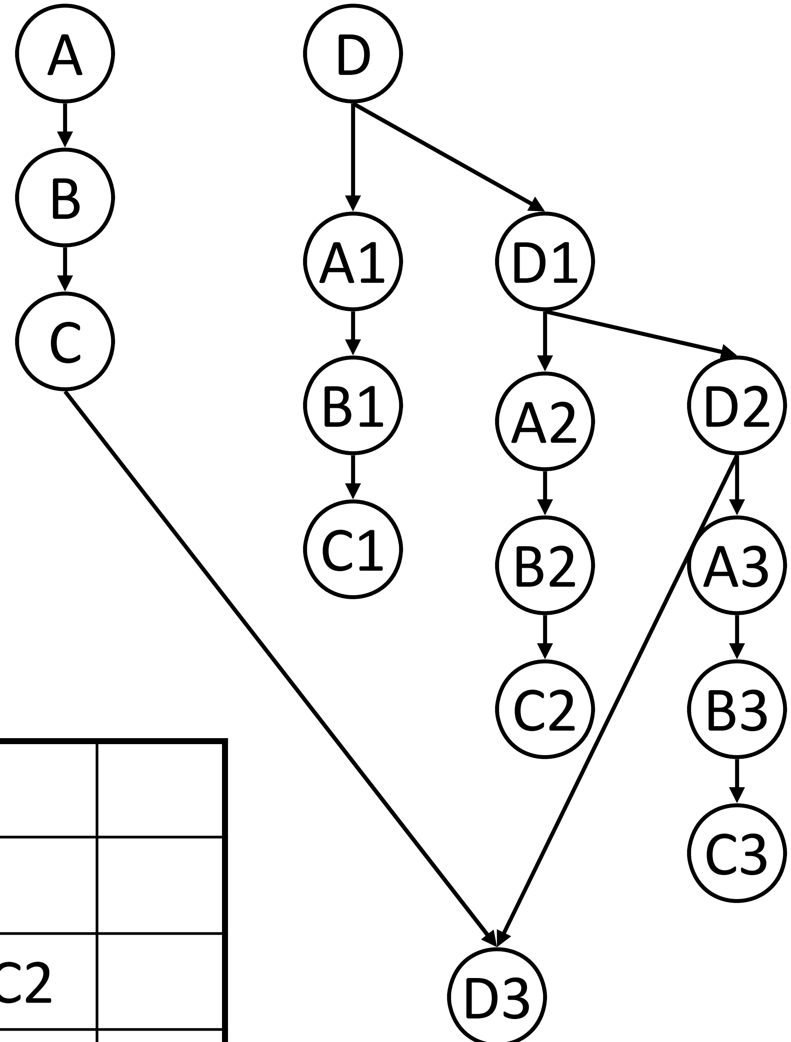
A: a ← ld [d]
B: b ← a * a
C: st [d], b
D: d1 ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1: st [d1], b1
D1: d2 ← d1 + 4
A2: a2 ← ld [d2]
B2: b2 ← a2 * a2
C2: st [d2], b2
D2: d ← d2 + 4



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

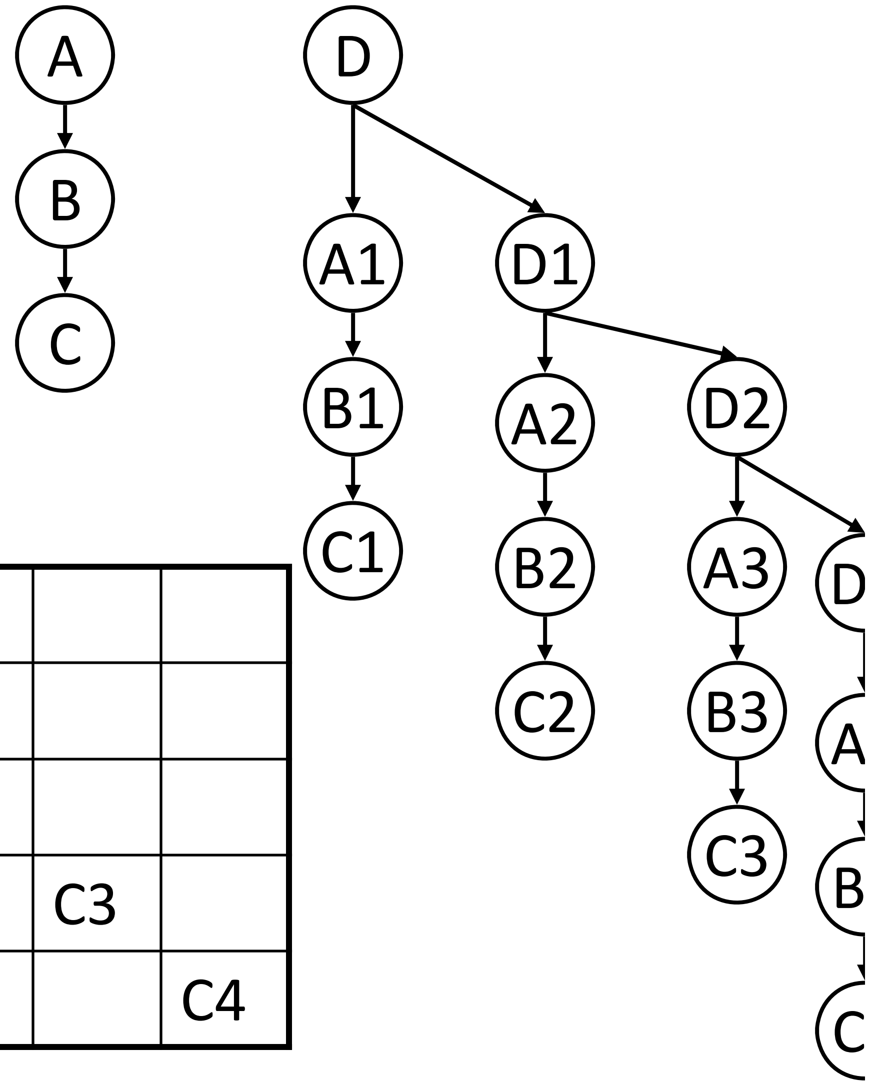
Unroll Some More

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



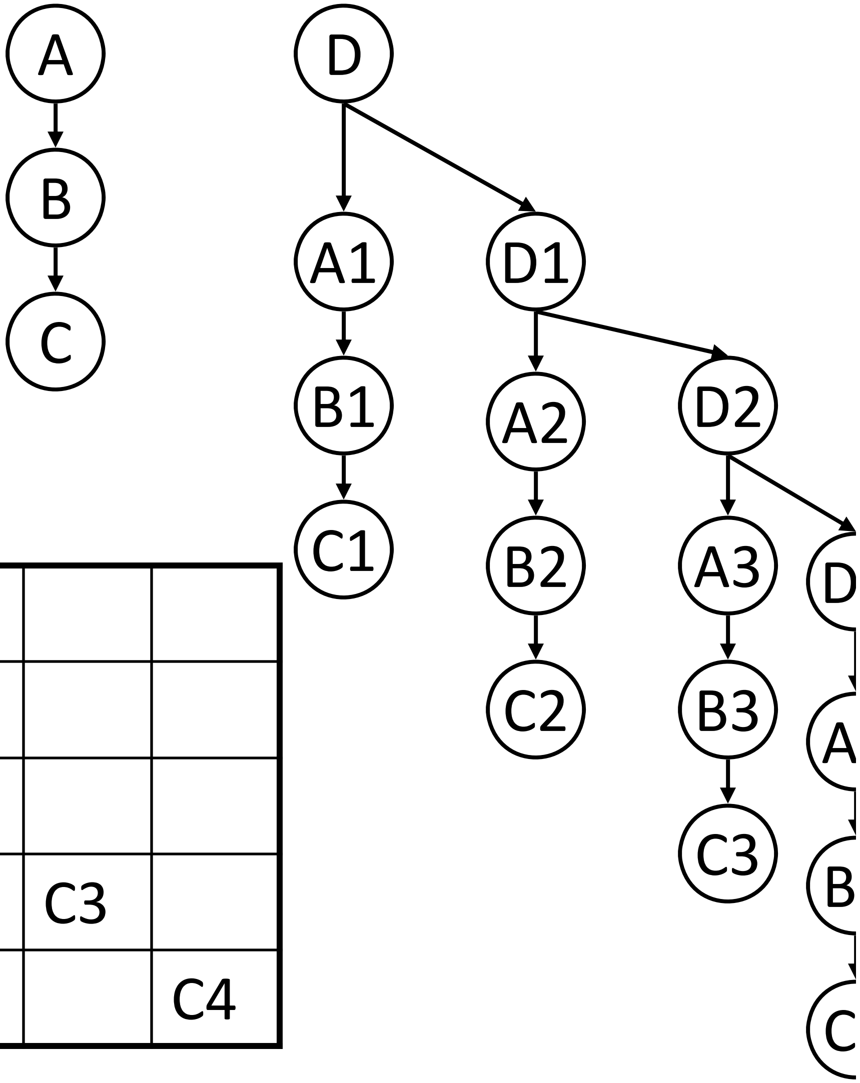
A		B		C		D3		
D		A1		B1		C1		
	D1		A2		B2		C2	
		D2		A3		B3		C3

One More Time



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

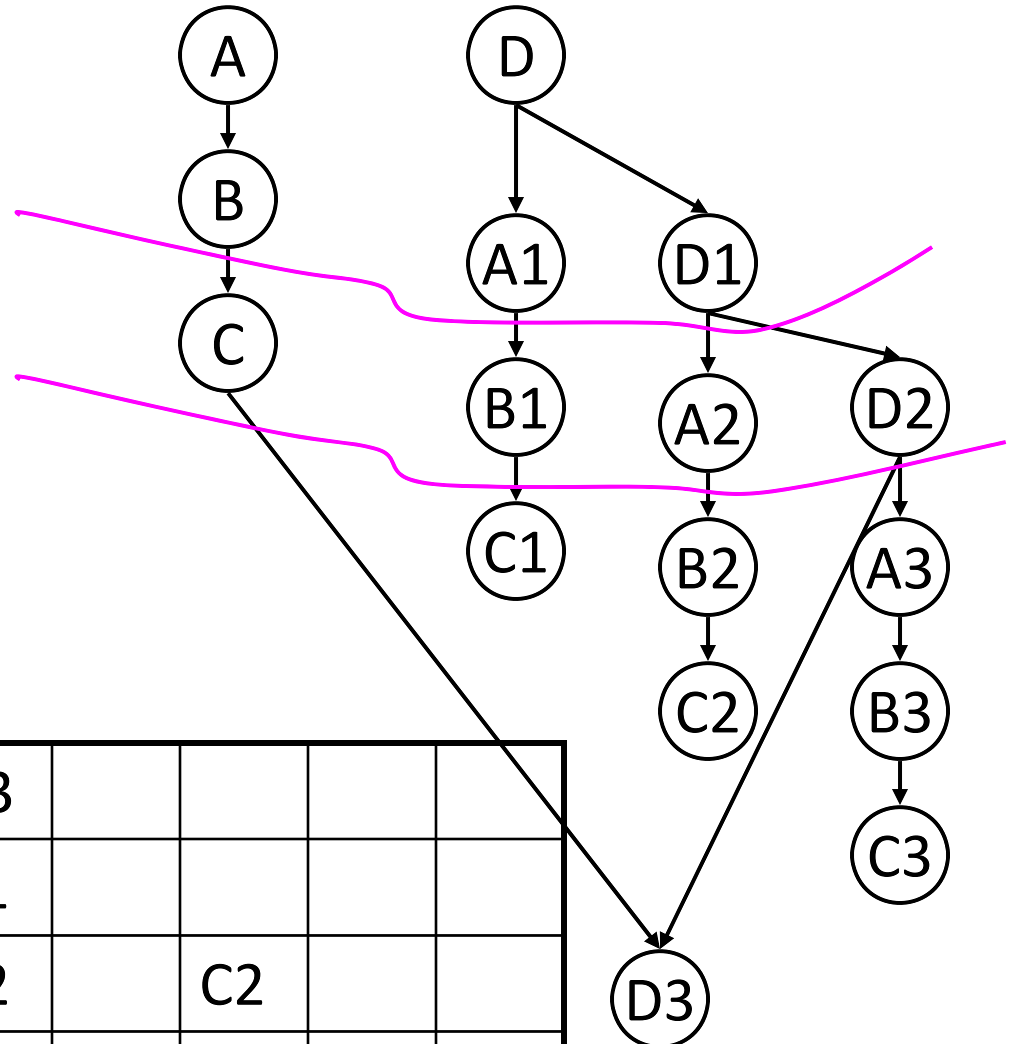
Can Rearrange



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

Rearrange

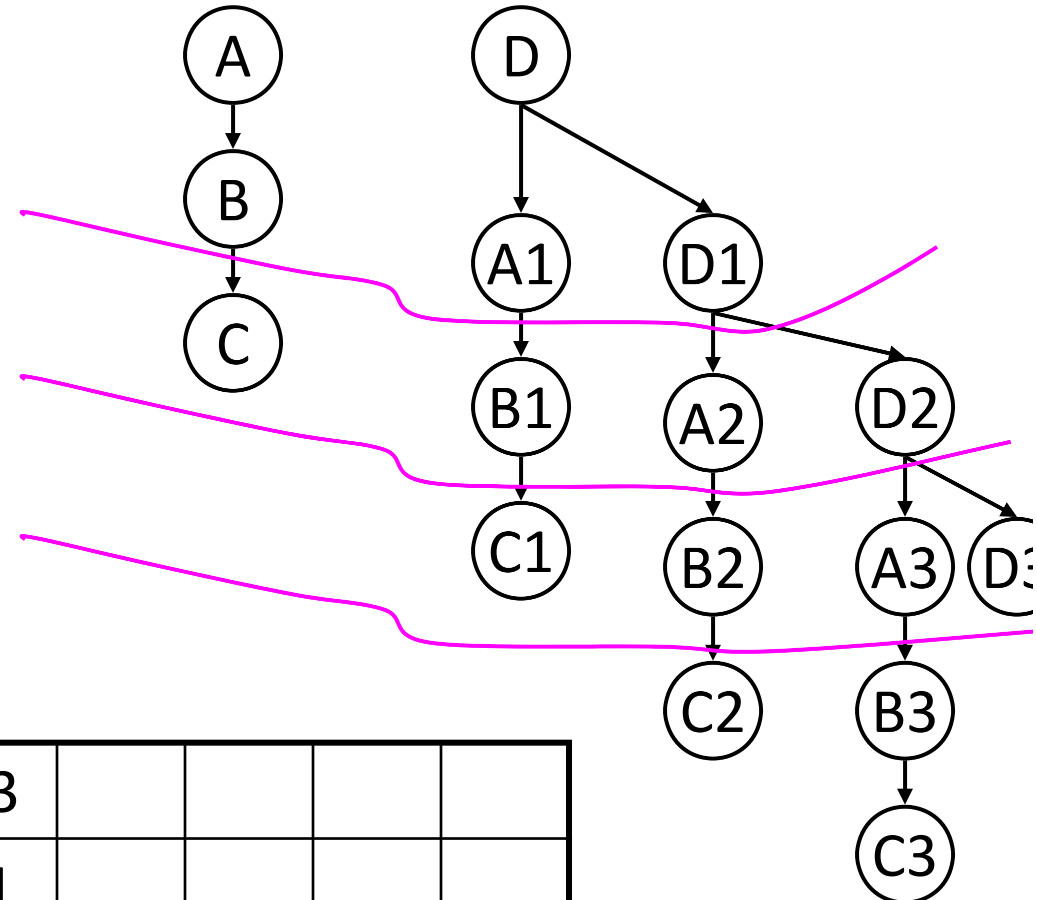
```
A:   a ←   ld [d]
B:   b ←   a * a
C:         st [d], b
D:   d1 ←  d + 4
A1:  a1 ←  ld [d1]
B1:  b1 ←  a1 * a1
C1:        st [d1], b1
D1:  d2 ←  d1 + 4
A2:  a2 ←  ld [d2]
B2:  b2 ←  a2 * a2
C2:        st [d2], b2
D2:  d ←   d2 + 4
```



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

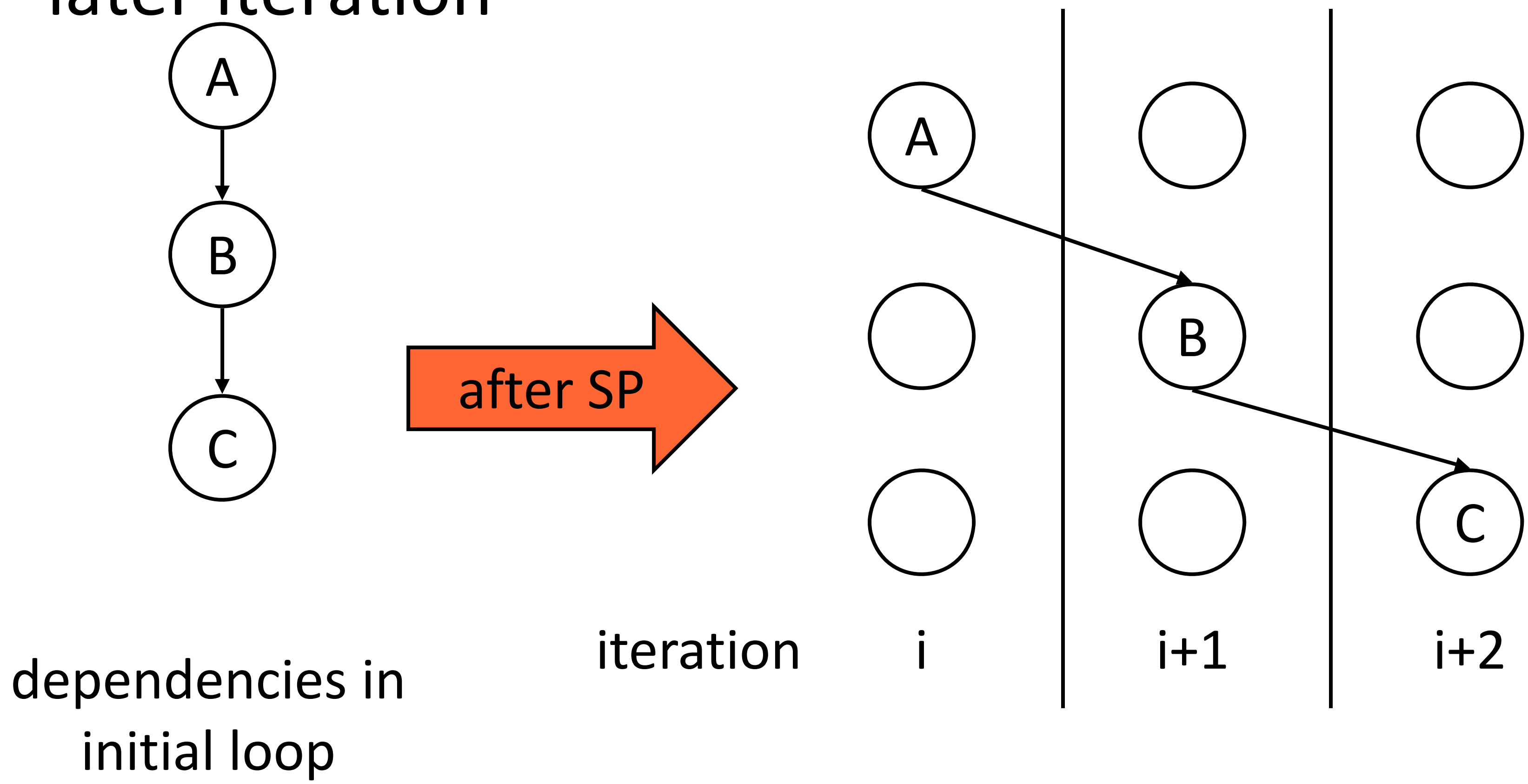
SP Loop

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

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

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

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

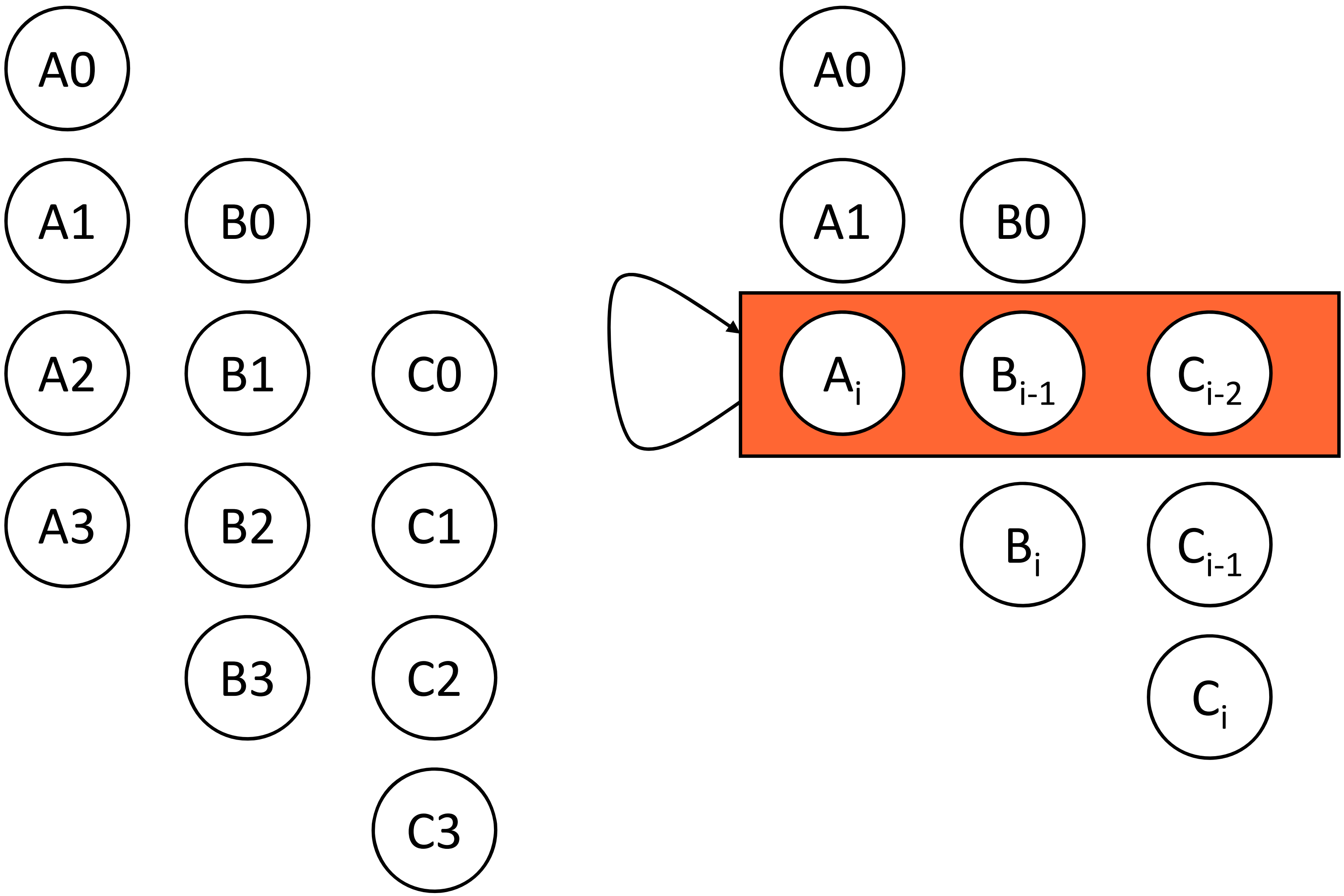
Goal of SP

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



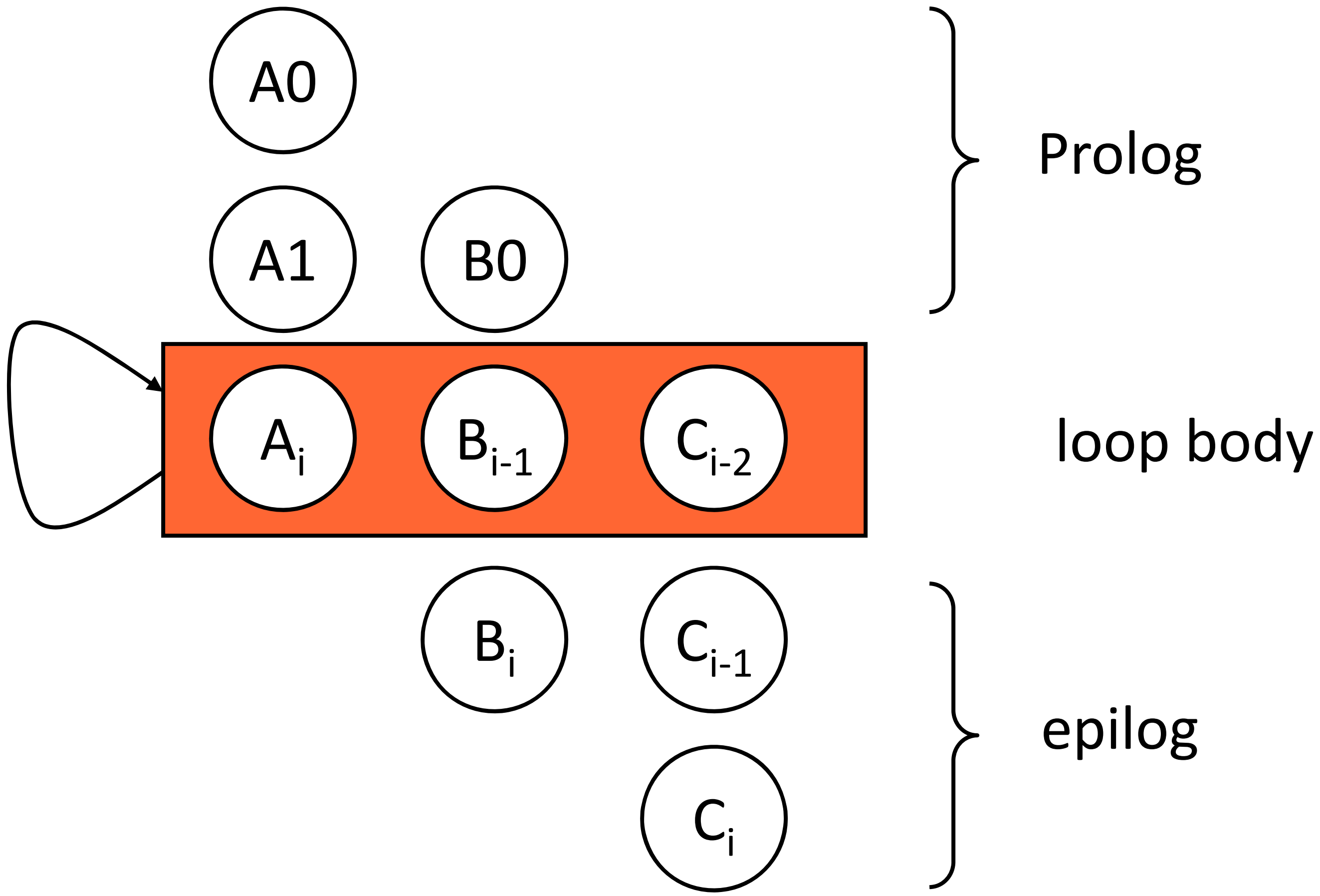
Example

Assume operating on a infinite wide machine



Example

Assume operating on a infinite wide machine



Dealing with exit conditions

```
for (i=0; i<N; i++)
{
    A_i
    B_i
    C_i
}
```

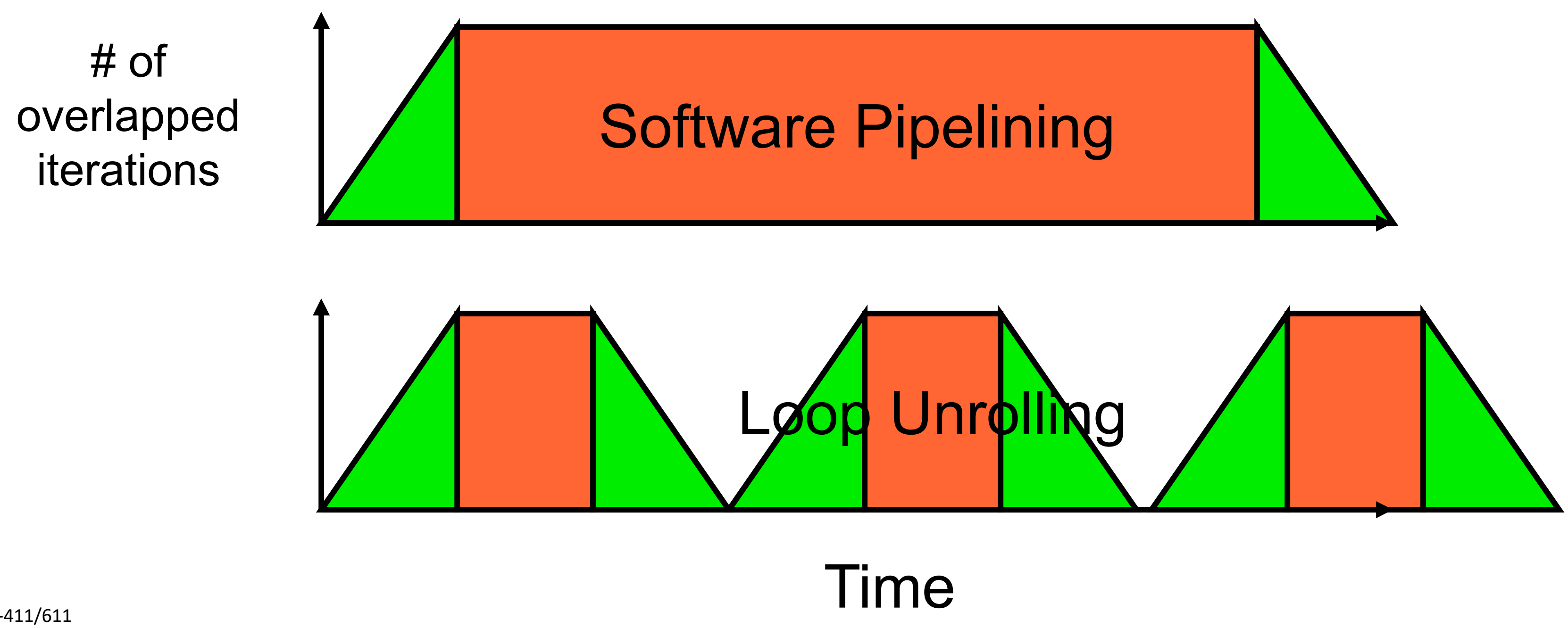
```
i=0
if (i >= N) goto done
A_0
B_0
if (i+1 == N) goto last
i=1
A_1
if (i+2 == N) goto epilog
i=2
```

```
loop:
    A_i
    B_i-1
    C_i-2
    i++
    if (i < N) goto loop
epilog:
    B_i
    C_i-1
last:
    c_i
done:
```

Loop Unrolling V. SP

For SuperScalar

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



Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

δ^f loop carried

δ^f loop independent

Unroll Loop to Find Dependencies

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

δ^f loop carried

δ^f loop independent

```
A[0] = B[0];
B[1] = A[0];
A[1] = B[1];
B[2] = A[1];
A[2] = B[2];
B[3] = A[2];
   ⋮
```

i=0

i=1

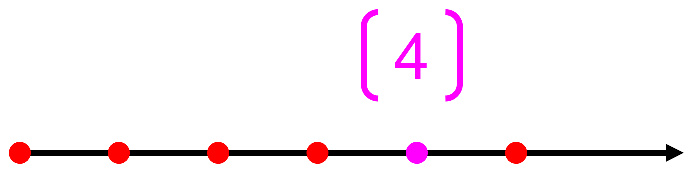
i=2

Distance/Direction of the dependence is also important.

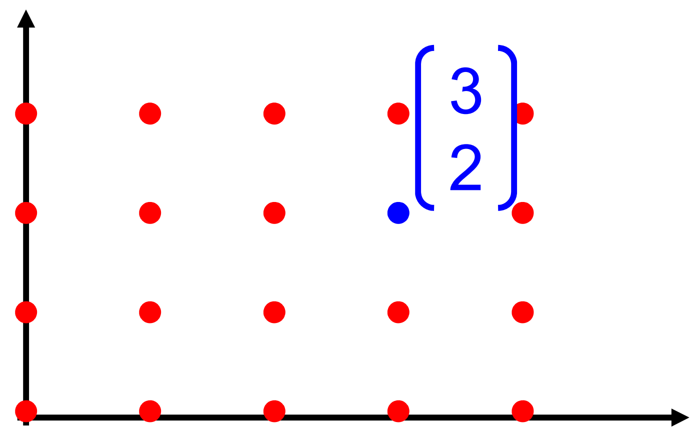
Iteration Space

Every iteration generates a point in an n-dimensional space, where n is the depth of the loop nest.

```
for (i=0; i<n; i++) {
   •••
}
```




```
for (i=0; i<n; i++)
   for (j=0; j<4; j++) {
      •••
}
```



Distance Vector

```
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

```
A[0] = B[0];
B[1] = A[0];   } i=0
A[1] = B[1];
B[2] = A[1];   } i=1
A[2] = B[2];
B[3] = A[2];   } i=2
     ⋮
```

Distance vector is the difference between the target and source iterations.

$$\mathbf{d} = \mathbf{I}_t - \mathbf{I}_s$$

Exactly the distance of the dependence, i.e.,

$$\mathbf{I}_s + \mathbf{d} = \mathbf{I}_t$$

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]
```

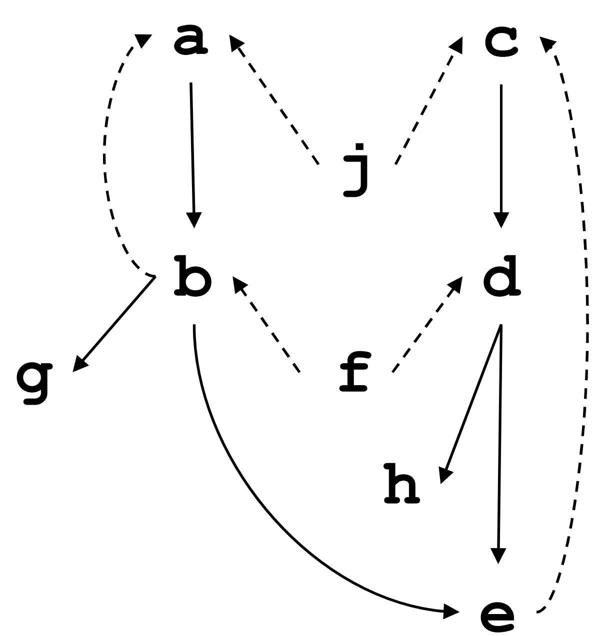
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

Aiken/Nicolau Scheduling
Step 2

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

DDG for rolled loop:

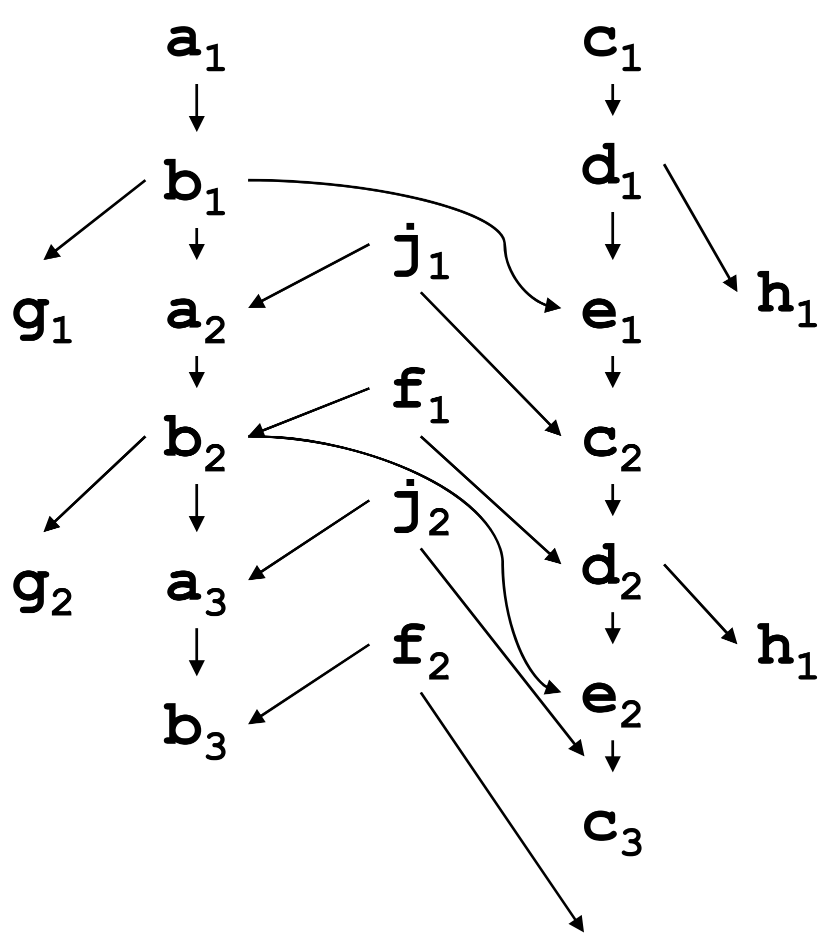
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```



Aiken/Nicolau Scheduling
Step 2, cont'd

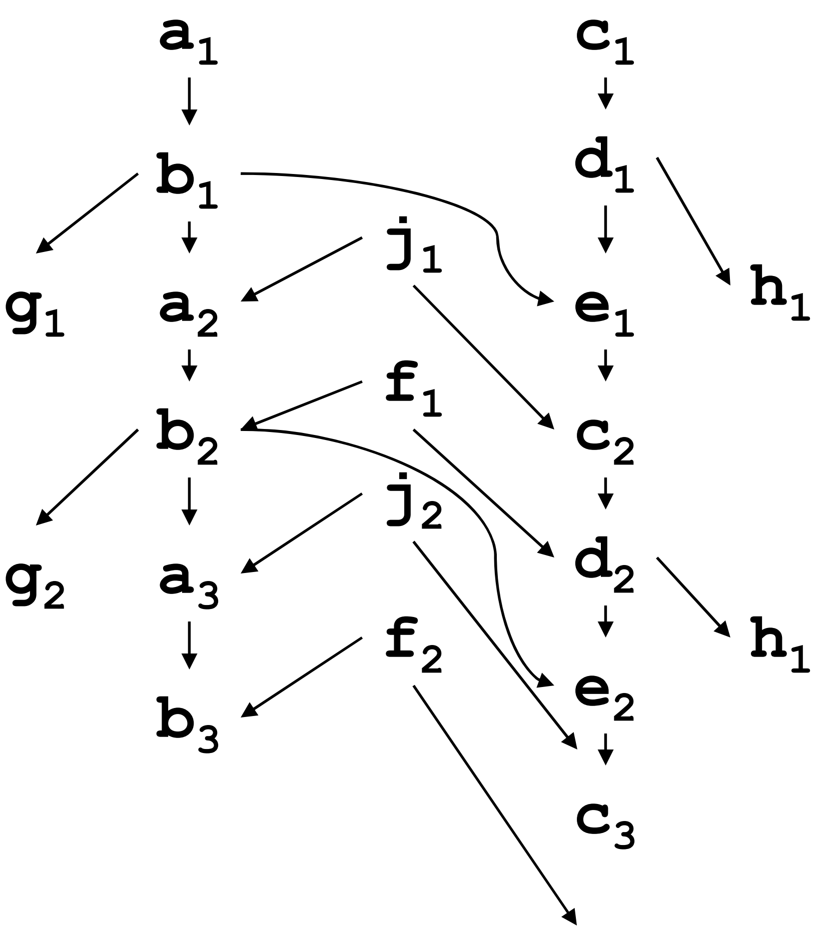
DDG for unrolled loop:

```
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 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 5

“Coalesce” the slopes.

iteration

cycle	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	

iteration

cycle	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				← Prologue/entry code
5		dg	a				
6		eh	b	fj			
7			cg	a			
8			d	b			
9			eh	g	fj		← Loop body
10				c	a		
11				d	b		
12				eh	g		
13					c		← Epilogue/exit code
14					d		
15					eh		

(Loop body box: cycles 8–10.)

Aiken/Nicolau Scheduling Step 7

Generate code.

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

$a1 := j0 \oplus b0$ $c1 := e0 \oplus j0$ f1 := U[1]   j1 := X[1]

$b1 := a1 \oplus f0$ $d1 := f0 \oplus c1$ f2 := U[2]   j2 := X[2]

$e1 := b1 \oplus d1$ V[1] := b1   W[1] := d1 $a2 := j1 \oplus b1$

$c2 := e1 \oplus j1$ $b2 := a2 \oplus f1$ f3 := U[3]   j3 := X[3]

$d2 := f1 \oplus c2$ V[2] := b2 $a3 := j2 \oplus b2$

$e2 := b2 \oplus d2$ W[2] := d2 $b3 := a3 \oplus f2$ f4 := U[4]   j4 := X[4]

$c3 := e2 \oplus j2$ V[3] := b3 $a4 := j3 \oplus b3$ i := 3

L:

$d_i := f_{i-1} \oplus c_i$ $b_{i+1} := a_i \oplus f_i$

$e_i := b_i \oplus d_i$ W[i] := d_i V[i+1] := b_{i+1} f_{i+2} := U[I+2] j_{i+2} := X[i+2]

$c_{i+1} := e_i \oplus j_i$ $a_{i+2} := j_{i+1} \oplus b_{i+1}$ i := i+1   if i<N-2 goto L

$d_{N-1} := f_{N-2} \oplus c_{N-1}$ $b_N := a_N \oplus f_{N-1}$

$e_{N-1} := b_{N-1} \oplus d_{N-1}$ W[N-1] := d_{N-1} v[N] := b_N

$c_N := e_{N-1} \oplus j_{N-1}$

$d_N := f_{N-1} + c_N$

$e_N := b_N \oplus d_N$ 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
```

Scalar Replacement

- Replaces subscripted array references with scalars.
- AKA: register pipelining
- Benefits:
  - Reduces memory traffic
  - Register allocation made possible
  - Easier to software pipeline

Example: MM

```
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      for (k=0; k<N; k++)
         C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

- replace C[][] with scalar in inner loop.
- Reduces memory references by $2(N^3-N^2)$

```
for (i=0; i<N; i++)
   for (j=0; j<N; j++) {
      sum = c[i][j];
      for (k=0; k<N; k++)
         sum = sum + A[i][k]*B[k][j];
      c[i][j] = sum;
   }
```

Scalar Replacement data structures

- Lets consider loops without conditionals
- Define the period of a loop carried dependence for edge e, p(e), as the CONSTANT number of iterations between the references at tail and head.
  (If not constant we can't do it).
- Build a partial dependence graph including
  - flow (R after W) and
  - input dependencies (R after R)

  And the dependencies
  - have a constant period
  - are:
    - loop independent or
    - carried by innermost loop

Scalar Replacement Alg

- For a period of p(e) cycles, use p(e)+1 temporaries
  t_0 to $t_{p(e)}$
- In body of loop:
  - Replace A[i] with t_0
  - Replace A[i+j] with t_j
- At end of innermost loop body add assignments
  $t_{p(e)} = t_{p(e)-1}; \ldots ; t_1 \leftarrow t_0$
- Init temps by peeling off p(e) iterations

Example: MM

```
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      for (k=0; k<N; k++)
         C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

p=<0,1>

- replace C[][] with scalar in inner loop.
- Reduces memory references by $2(N^3-N^2)$

```
for (i=0; i<N; i++)
   for (j=0; j<N; j++) {
      sum = c[i][j];
      for (k=0; k<N; k++)
         sum = sum + A[i][k]*B[k][j];
      c[i][j] = sum;
   }
```

Scalar Replacement: Loop Body

```
for (i=0; i<n; i++) {
   b[i+1] = b[i] + f
   a[i] = 2 * b[i] + c[i]
}
```

p=<1,0>

p=<0,1>

- We need two temporaries: t0, t1
- Replace b[i] with t0 and b[i+1] with t1
- Insert copies at bottom of loop

```
for (i=0; i<n; i++) {
   t1 = t0 + f
   b[i+1] = t1
   a[i] = 2 * t0 + c[i]
   t0 = t1
}
```

Scalar Replacement: Init

```
for (i=0; i<n; i++) {
    t1 = t0 + f
    b[i+1] = t1
    a[i] = 2 * t0 + c[i]
    t0 = t1
}
```

1) Peel of p(e) iterations of loop

```
b[1] = b[0] + f
a[0] = 2 * b[0] + c[0]
```

2) after replacement

```
t0 = b[0]
t1 = t0 + f
b[1] = t1
a[0] = 2 * t0 + c[0]
```

3) If we aren't sure of trip count

```
if (n>=0) {
    t0 = b[0]
    t1 = t0 + f
    b[1] = t1
    a[0] = 2 * t0 + c[0]
}
```

Finished

```
for (i=0; i<n; i++) {
   b[i+1] = b[i] + f
   a[i] = 2 * b[i] + c[i]
}
```

```
if (n>=0) {
   t0 = b[0]
   t1 = t0 + f
   b[1] = t1
   a[0] = 2 * t0 + c[0]
}
for (i=1; i<n; i++) {
   t1 = t0 + f
   b[i+1] = t1
   a[i] = 2 * t0 + c[i]
   t0 = t1
}
```

Back to 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

Resource Constraints

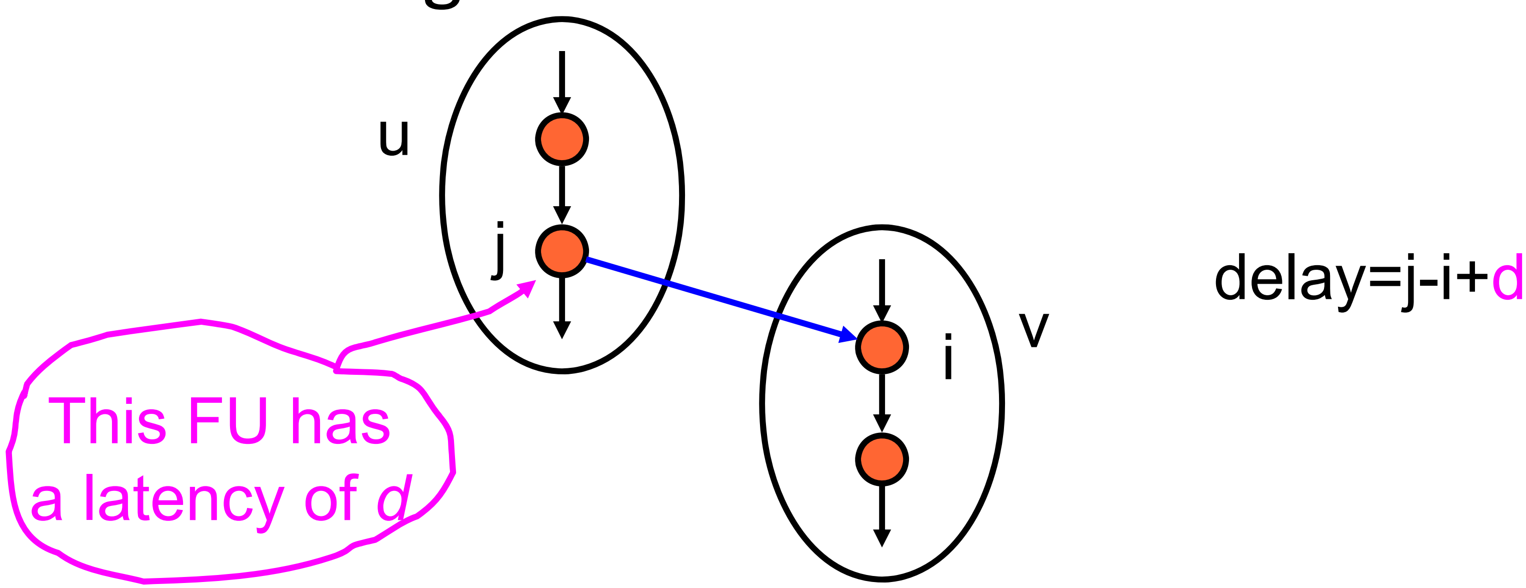
- Minimally indivisible sequences, *i* and *j*, can execute together if combined resources in a step do not exceed available resources.
- R(i) is a resource configuration vector

  R(i) is the number of units of resource i
- r(i) is a resource usage vector s.t.

  $0 \leq r(i) \leq R(i)$
- Each node in G has an associated r(i)

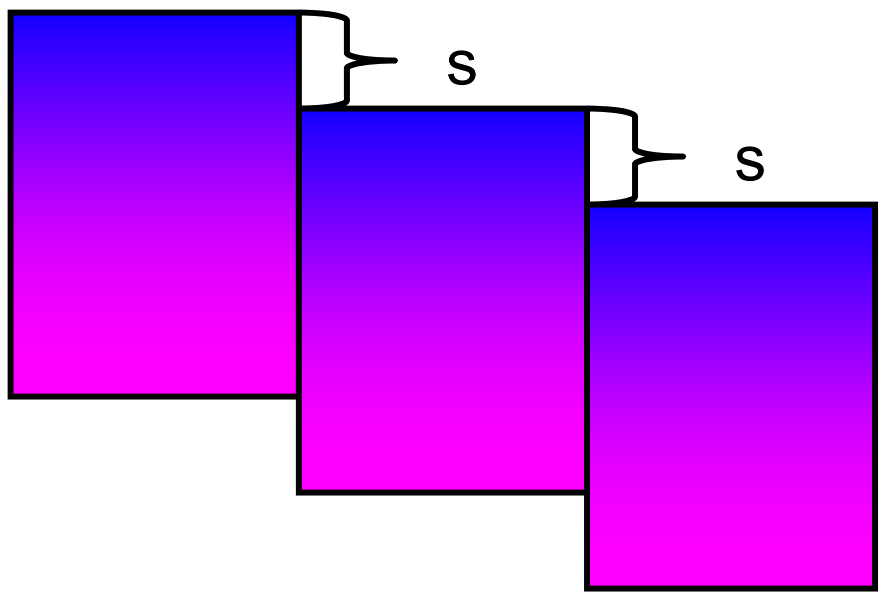
Precedence Constraints

- Data Dependence + Latency of the functional unit being used
- The precedence constraint between two nodes, u and v, is the minimal delay between starting u and v in the schedule.



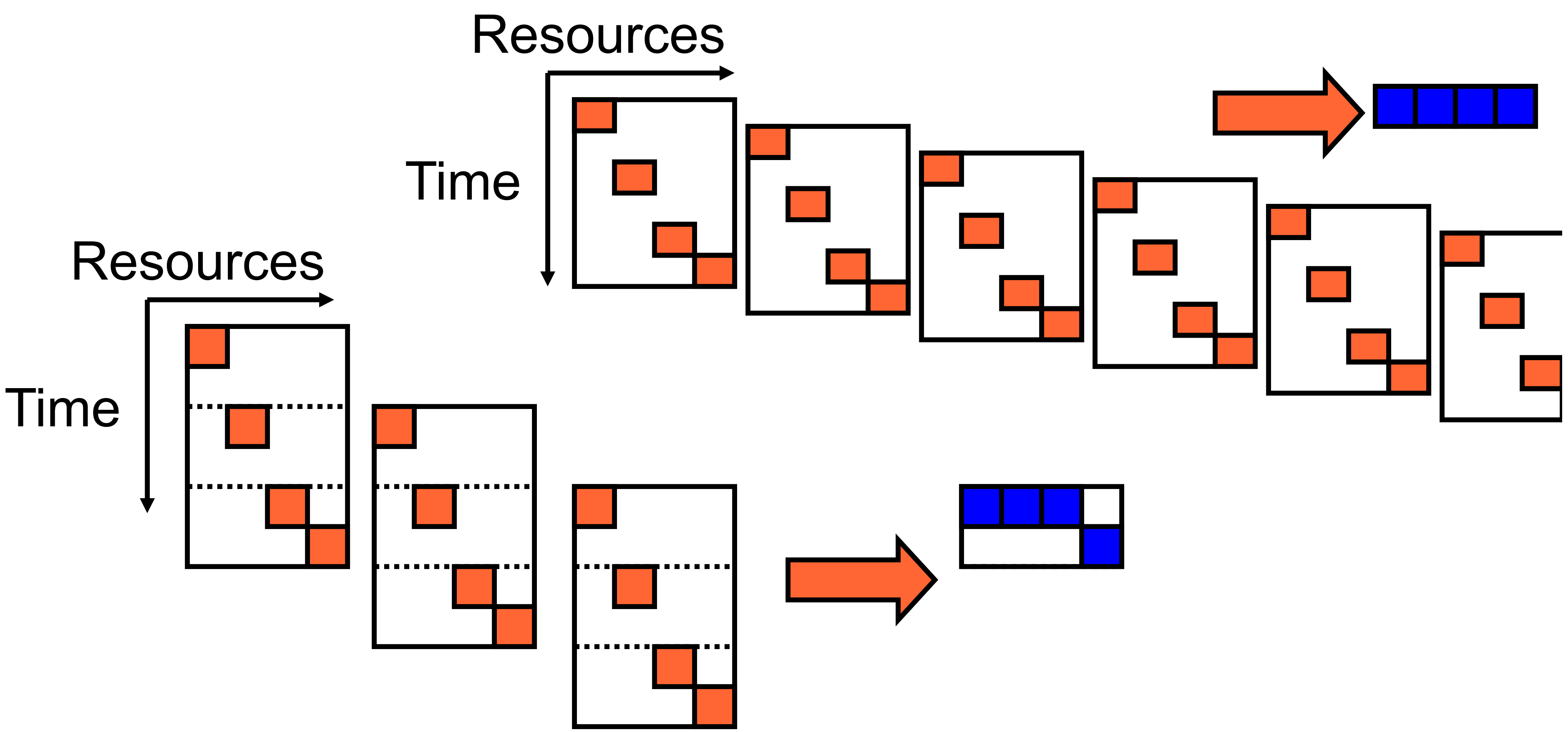
Software Pipelining Goal

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



Modulo Resource Constraints

- Combine the resource constraints of instructions at steps i,i+s,i+2s,i+3s, etc.



Precedence Constraints

- 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 loop
  - independent p=0
  - loop-carried p>0

Iterative Approach

- 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

Lower Bounds

- Resource Constraints: S_R

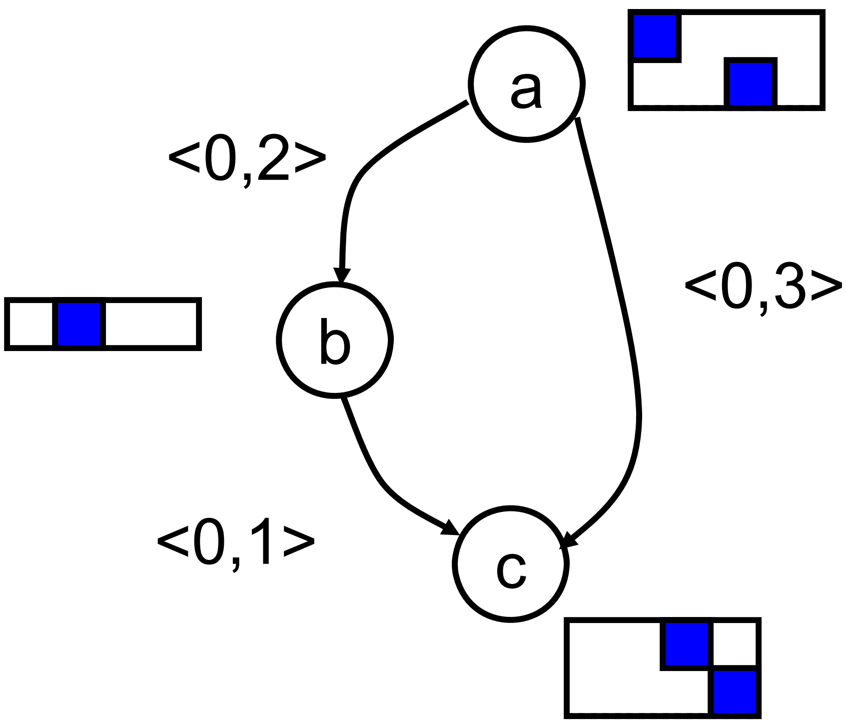
  maximum over all resources of # of uses divided by # available

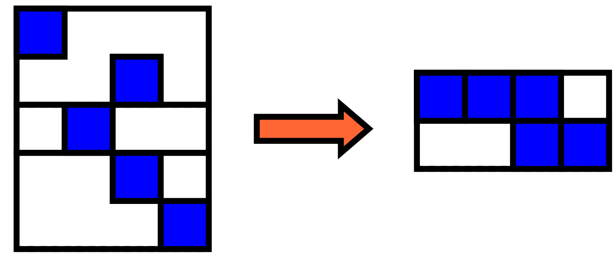
What is lower bound.
Is it tight?

- Precedence Constraints: S_E

  max over all cycles: d(c)/p(c)

Acyclic Example



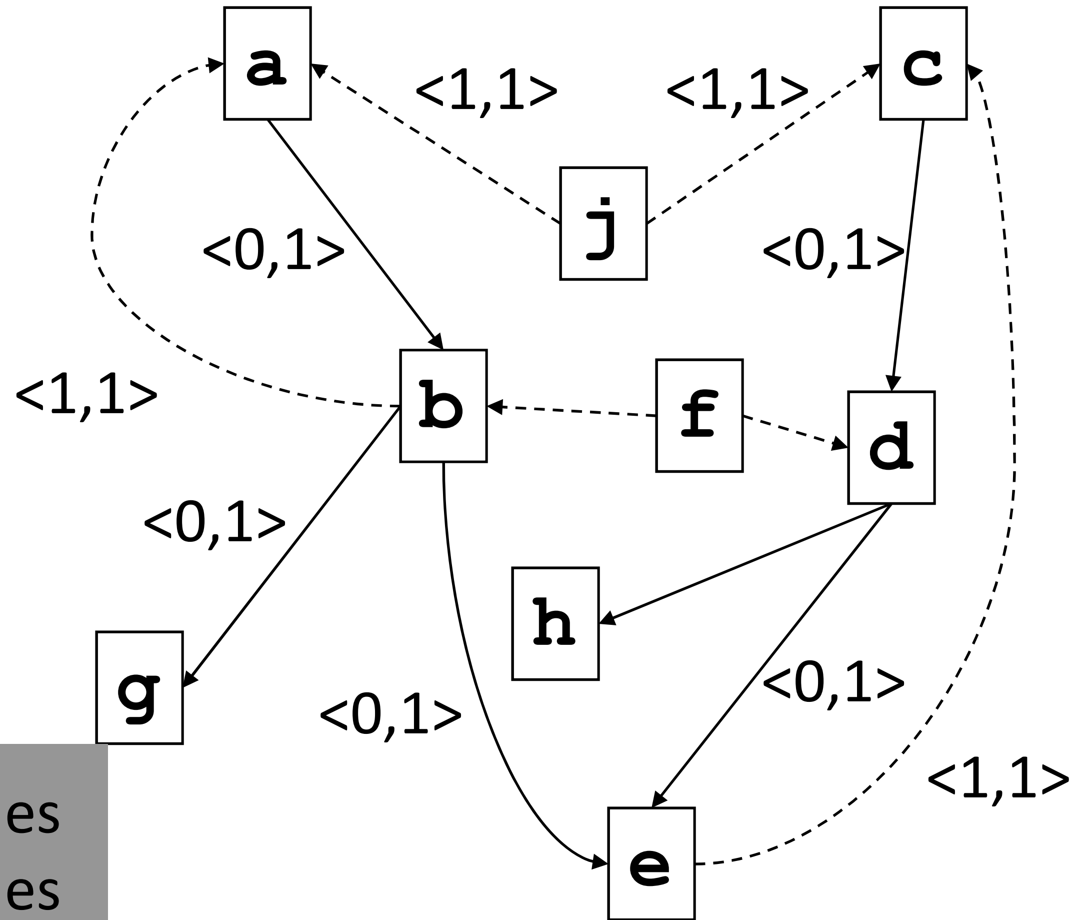


Lower Bound: S_R=2
Upper Bound: 5

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
  - 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)-sp(x,n))
- try and schedule n from earliest to earliest+s-1 s.t. resource constraints are obeyed.
- If we fail, then this schedule is faulty

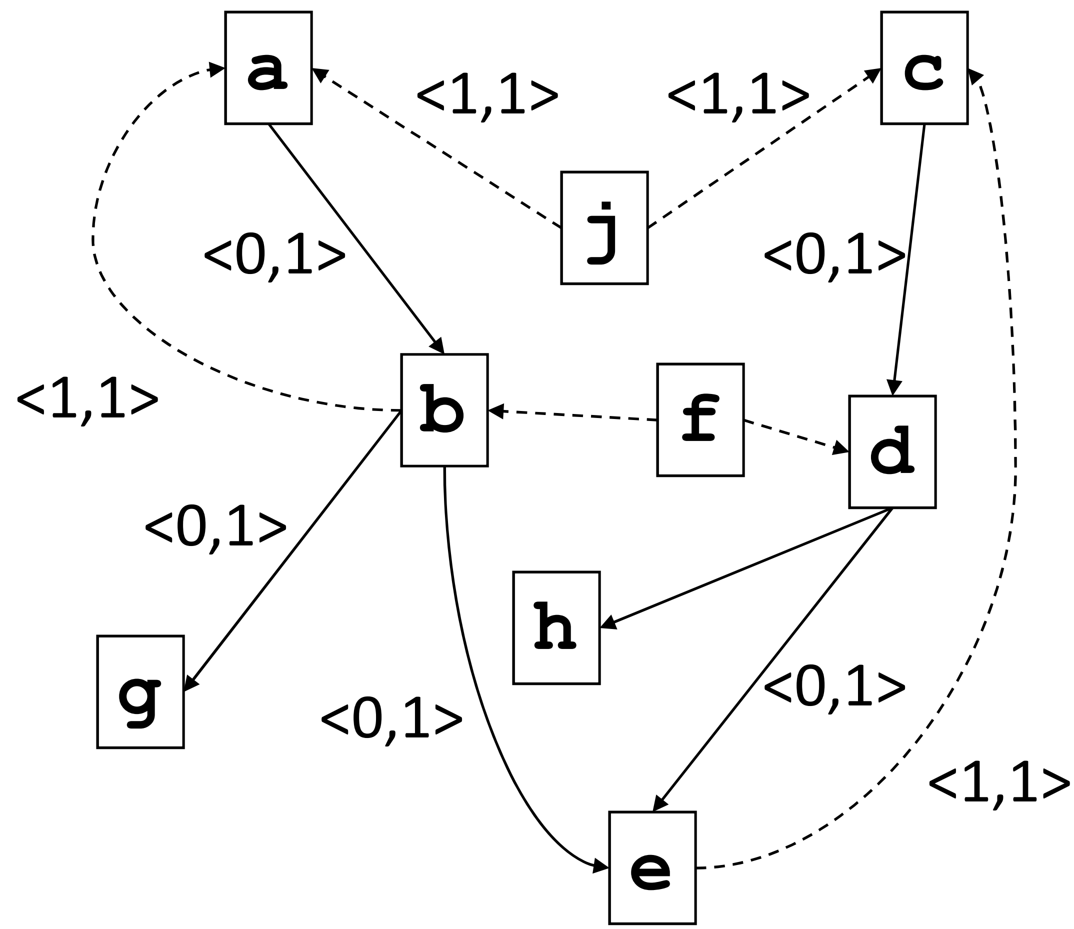
Scheduling algorithm – cont.

- 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)-sp(n,x), otherwise they are removed.
  - All schedule instructions (except n) that have data dependence conflicts are removed.
- repeat this some number of times until either
  - succeed, then register allocate
  - fail, then increase s

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]
```

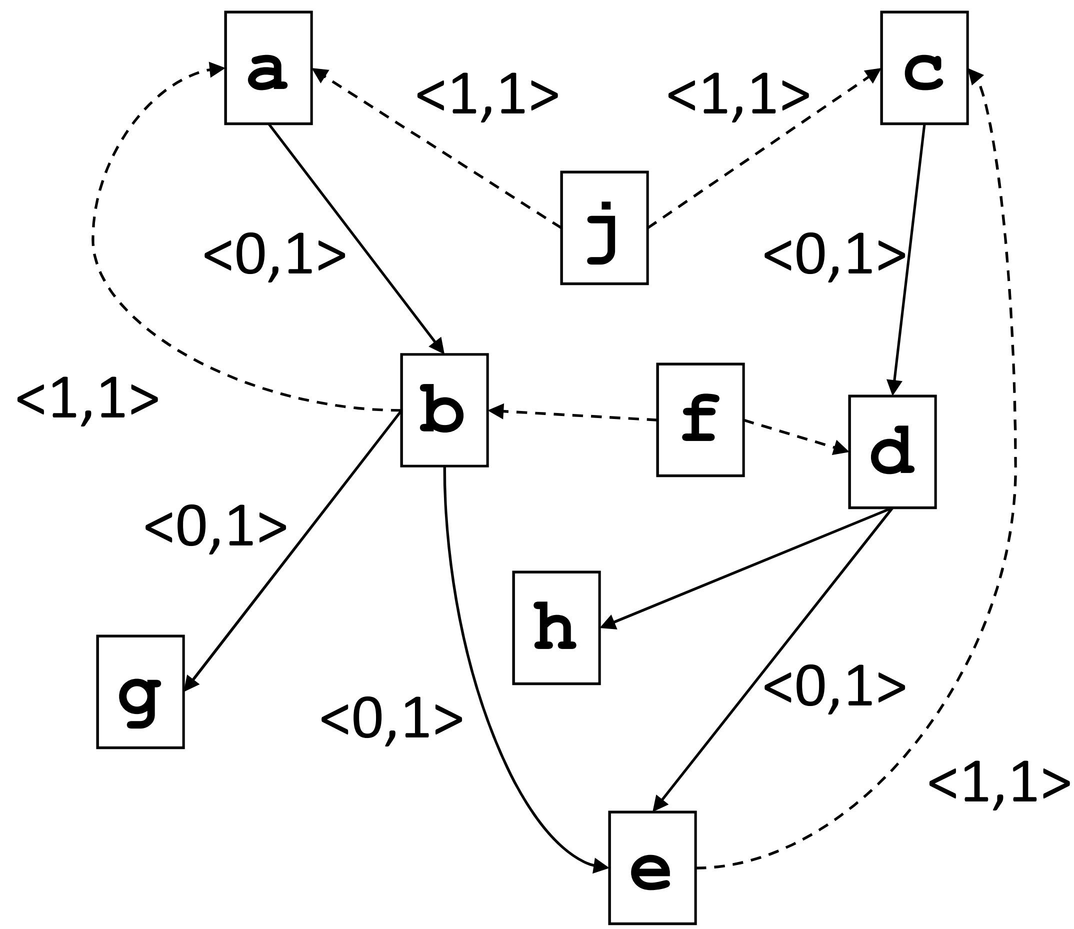
Priorities: ?



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]
```

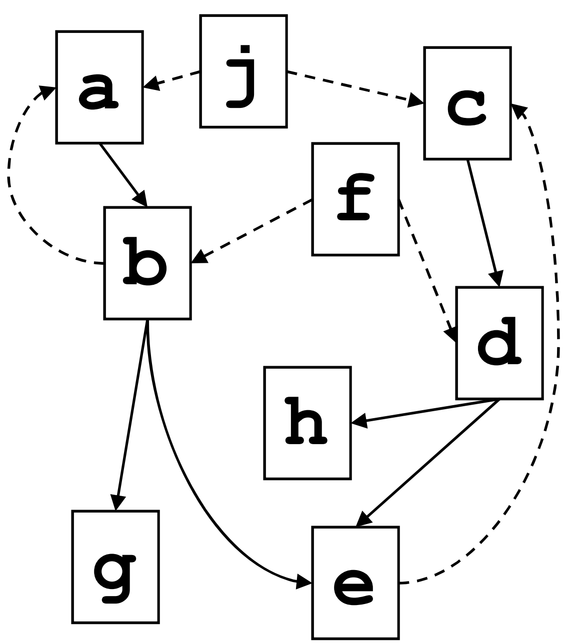
Priorities: c,d,e,a,b,f,j,g,h



```
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



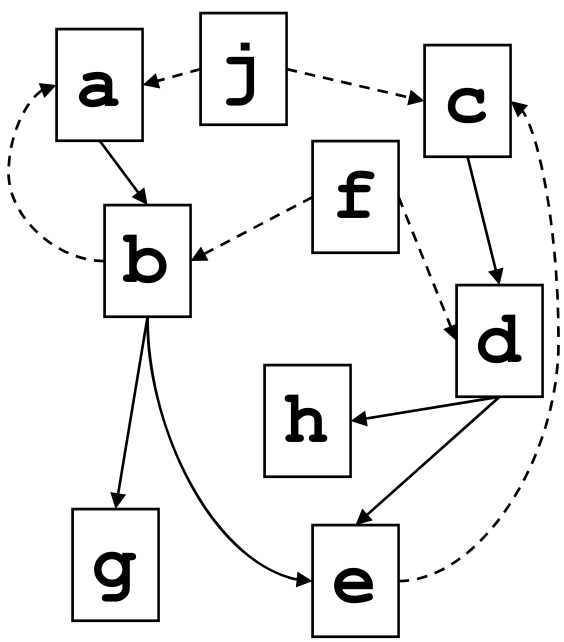
ALU	MU

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



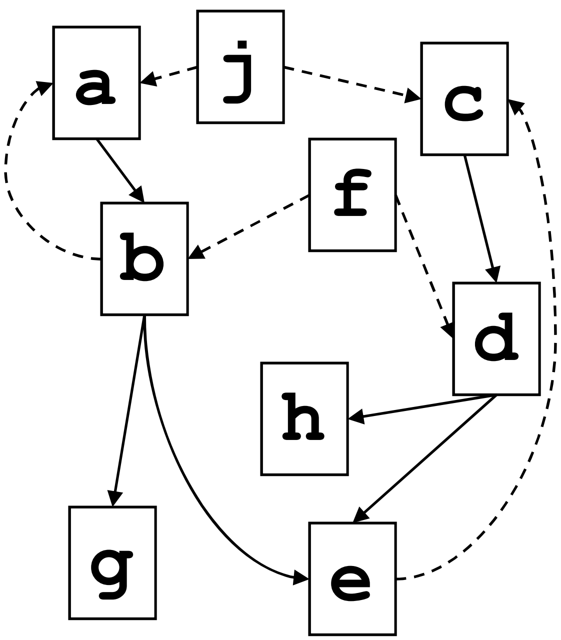
ALU	MU
c	
d	
e	

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



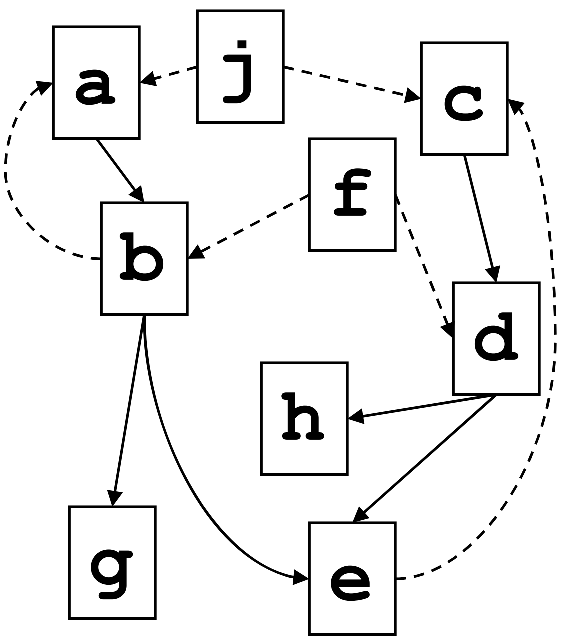
ALU	MU
c	
d	
e	
a	

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

Priorities: b,f,j,g,h



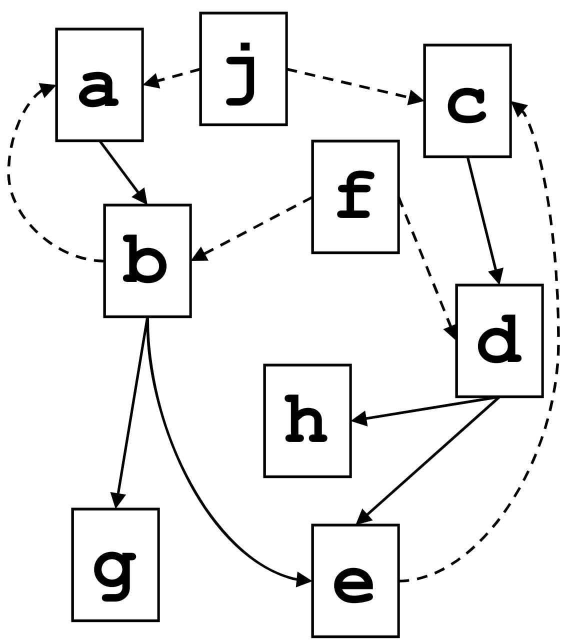
ALU	MU
c	
d	
e	
a	
b	

instr	σ
a	3
b	4
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: e,f,j,g,h



ALU	MU
c	
d	
a	
b	

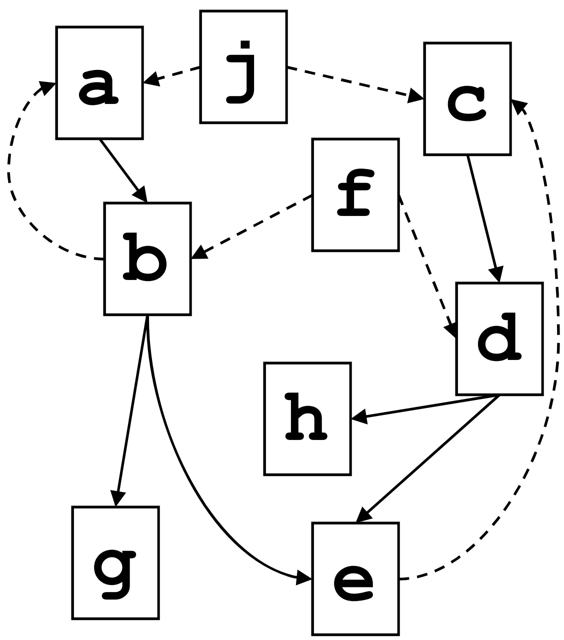
instr	σ
a	3
b	4
c	0
d	1
e	
f	
g	
h	
j	

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

Priorities: e,f,j,g,h



ALU	MU
c	
d	
e	
a	
b	

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

e causes e->c edge violation

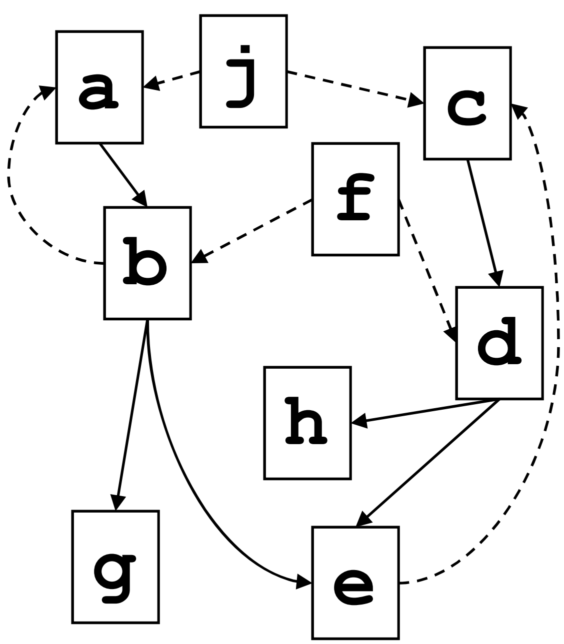
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

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	

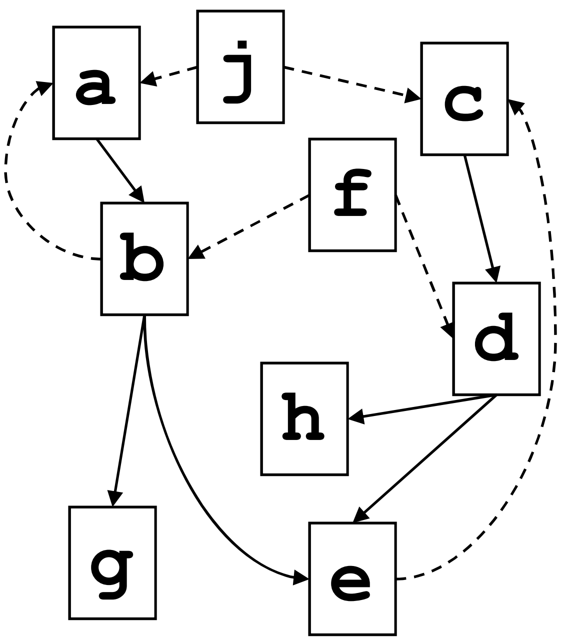
Priorities: f,j,g,h



```
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



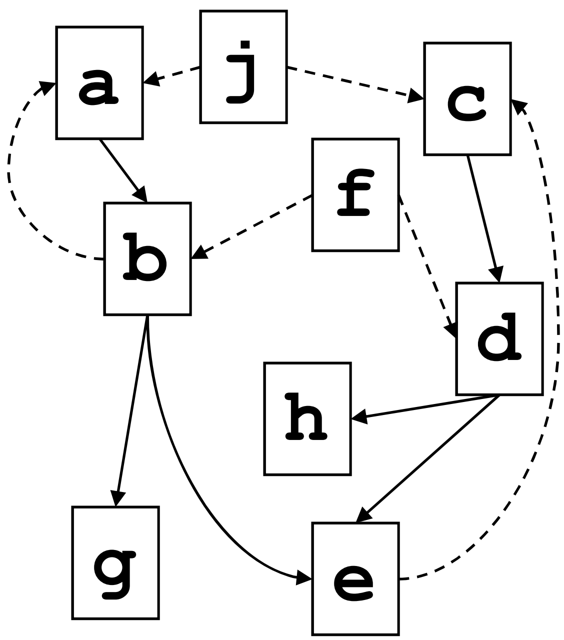
ALU	MU
c	f
d	j
e	
a	
b	

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

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