

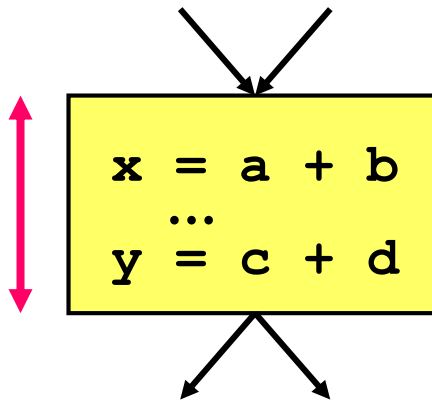
Lecture 21

Software Pipelining & Prefetching

- I. Software Pipelining
- II. Software Prefetching (of Arrays)
- III. Prefetching via Software Pipelining

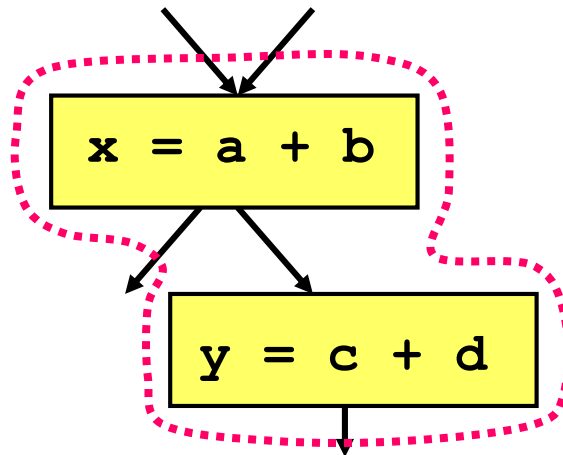
[ALSU 10.5, 11.11.4]

Scheduling Roadmap



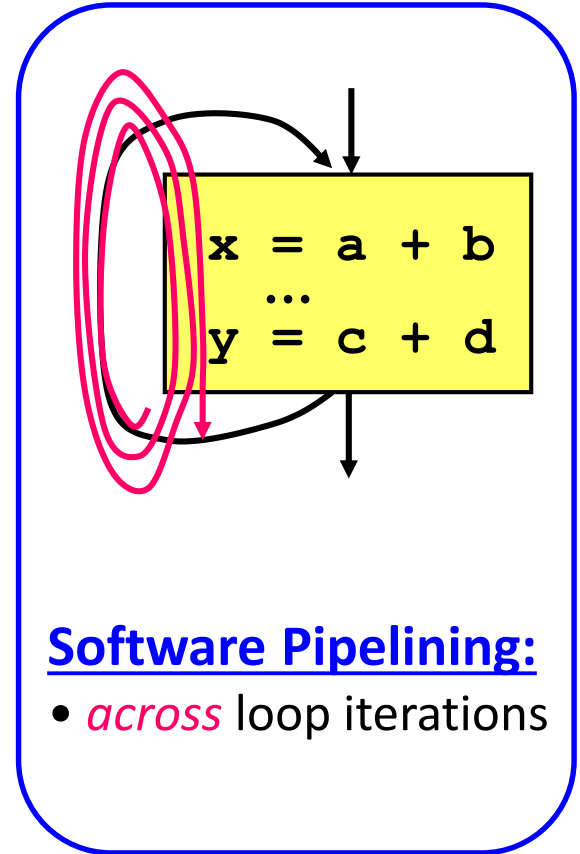
List Scheduling:

- *within* a basic block



Global Scheduling:

- *across* basic blocks



Software Pipelining:

- *across* loop iterations

Example of DoAll Loops

- **Machine:**
 - Per clock: 1 load, 1 store, 1 (2-stage) arithmetic op, with hardware loop op and auto-incrementing addressing mode.

- **Source code:**

```
for i = 0 to n-1
    D[i] = A[i]*B[i] + c
```

Figure 10.17
in ALSU

- **Code for one iteration:**

```
1. LD  R5,0(R1++)
2. LD  R6,0(R2++)
3. MUL R7,R5,R6
4.
5. ADD R8,R7,R4
6.
7. ST  0(R3++),R8
```

Initially:

R1 holds &A,
R2 holds &B,
R3 holds &D,
R4 holds c

- **Little or no parallelism within basic block**

Loop Unrolling

1.	L:	LD			
2.		LD			
3.			LD		
4.	MUL		LD		
5.		MUL		LD	
6.	ADD			LD	
7.		ADD			LD
8.	ST		MUL		LD
9.		ST			MUL
10.			ADD		
11.				ADD	
12.			ST		
13.				ST	BL (L)

Schedule after unrolling by a factor of 4



Figure 10.18
in ALSU

- Let **u** be the **degree of unrolling**, for **u** even:
 - Length of **u** iterations = $7+2(u-1)$
 - Execution time per source iteration = $(7+2(u-1)) / u = 2 + 5/u$

Software Pipelined Code

1.	LD				
2.	LD				
3.	MUL	LD			
4.		LD			
5.		MUL	LD		
6.	ADD		LD		
7.	L:		MUL	LD	
8.	ST	ADD		LD	BL (L)
9.				MUL	
10.		ST	ADD		
11.					
12.			ST	ADD	
13.					
14.				ST	

Figure 10.20
in ALSU

Execution time per source
iteration approaches 2

- Unlike unrolling, software pipelining can give optimal result with small code size blowup
- Locally compacted code may not be globally optimal
- **DOALL**: Can fill arbitrarily long pipelines with infinitely many iterations

Example of DoAcross Loop

Machine:

- Per clock: 1 load, 1 store, 1 (2-stage) arithmetic op fully pipelined, with hardware loop op and auto-incrementing addressing mode.

Loop:

```
Sum = Sum + A[i];  
B[i] = A[i] * c;
```



```
1. LD // A[i]  
2. MUL // A[i]*c  
3. ADD // Sum += A[i]  
4. ST // B[i]
```

Software Pipelined Code

```
1. LD  
2. MUL  
3. L: ADD LD  
4. ST MUL BL (L)  
5. ADD  
6. ST
```

Doacross loops

- Recurrences can be parallelized
- Harder to fully utilize hardware with large degrees of parallelism

Problem Formulation

Goals:

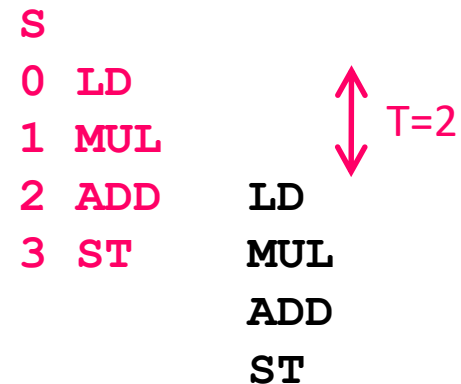
- maximize throughput
- small code size

Find:

- an **identical relative schedule $S(n)$** for every iteration
- a **constant initiation interval (T)**

such that

- the **initiation interval is minimized**



Complexity:

- NP-complete in general

Impact of Resources on Bound on Initiation Interval

- Example: Resource usage of 1 iteration
 - Machine can execute 1 LD, 1 ST, 2 ALU per clock
 - LD 1 cycle, MUL 3 cycles, ADD 2 cycles, fully pipelined

LD, LD, MUL, ADD, ST

- Lower bound on initiation interval?

for all resource i ,

number of units required by one iteration: n_i

number of units in system: R_i

Lower bound due to resource constraints: $\max_i \left\lceil \frac{n_i}{R_i} \right\rceil$

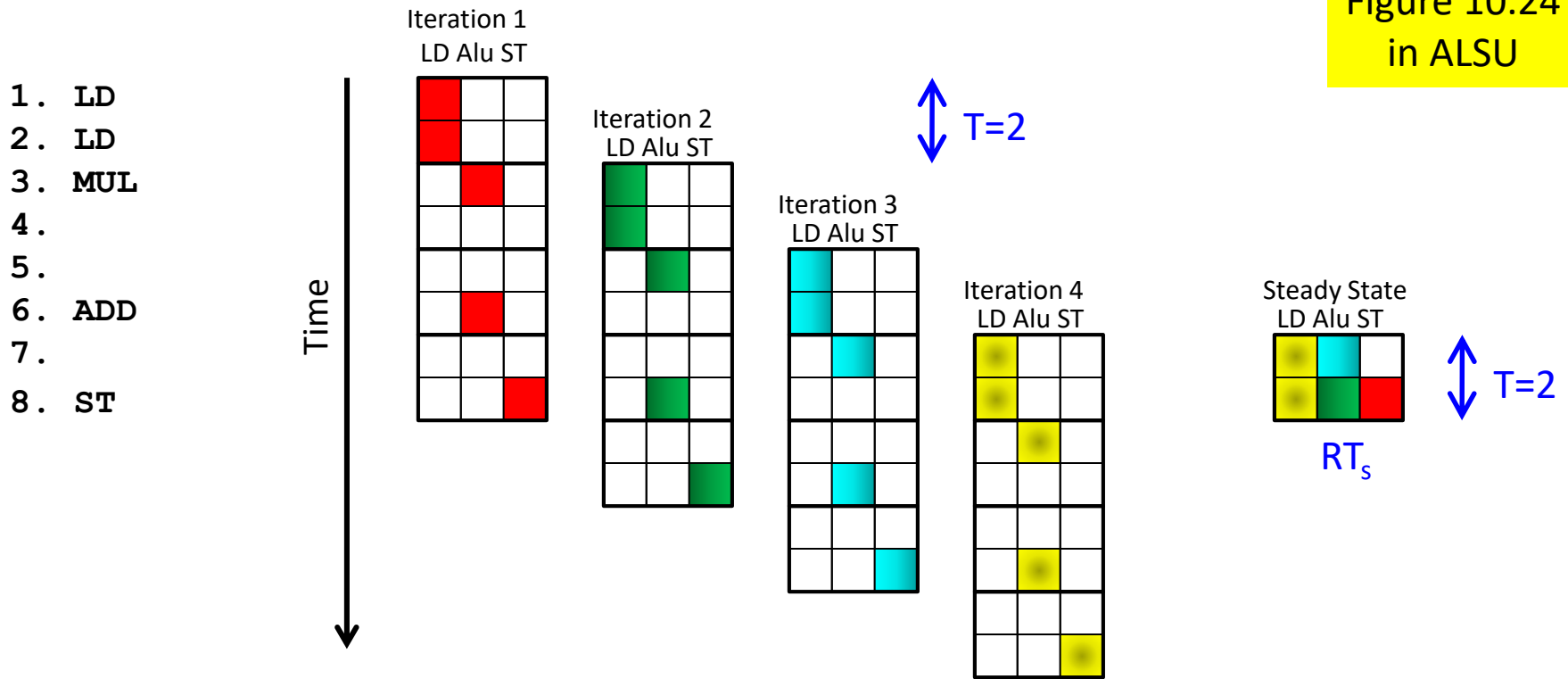
```
1. LD  R5, 0 (R1++)
2. LD  R6, 0 (R2++)
3. MUL R7, R5, R6
4.
5.
6. ADD R8, R7, R4
7.
8. ST  0 (R3++), R8
```

Lower bound?

2, due to LD

Scheduling Constraints: Resources

Figure 10.24
in ALSU

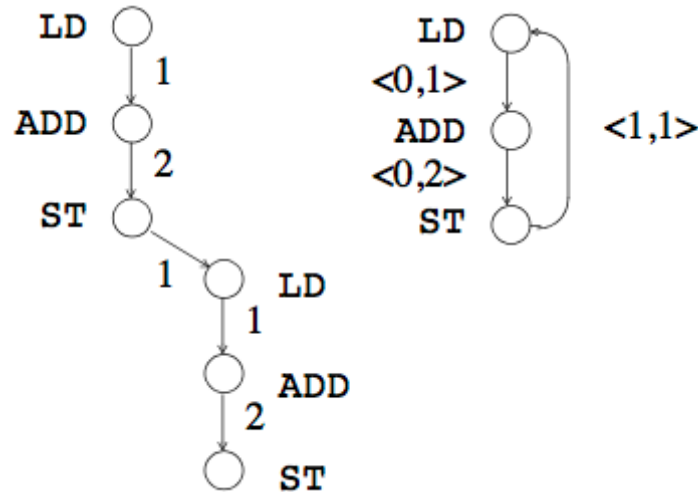


- RT : resource reservation table for single iteration
- RT_s : modulo resource reservation table (steady state)

$$RT_s[i] = \sum_{t|(t \bmod T = i)} RT[t]$$

Scheduling Constraints: Precedence

```
for (i = 0; i < n; i++) {
    *(p++) = *(q++) + c;
}
```



S(0) LD
 S(1) ADD
 S(2)
 S(3) ST

LD
 ADD
 ST

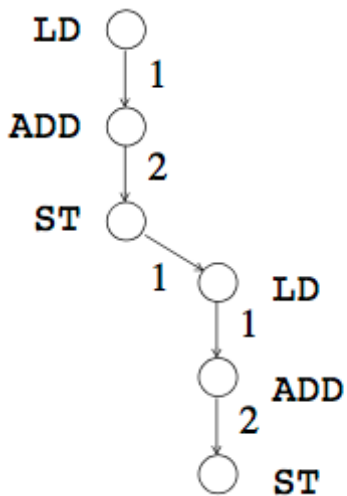
- Minimum initiation interval T? $1+2+1 = 4$
- $S(n)$: schedule for n with respect to the beginning of the schedule
- Label edges with $\langle \delta, d \rangle$
 - δ = iteration difference, d = delay

$$\delta \times T + S(n_2) - S(n_1) \geq d$$

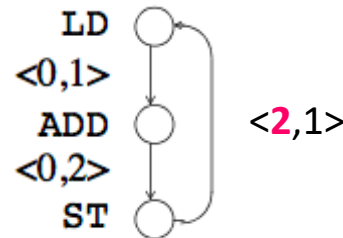
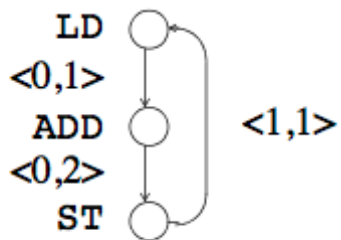
Minimum Initiation Interval

For all cycles c ,

$$T = \max_c \text{CycleLength}(c) / \text{IterationDifference}(c)$$



$$T = 4/1 = 4$$



$$A[i] = A[i-2] + B[j]$$

$$T = ? \quad 4/2 = 2$$

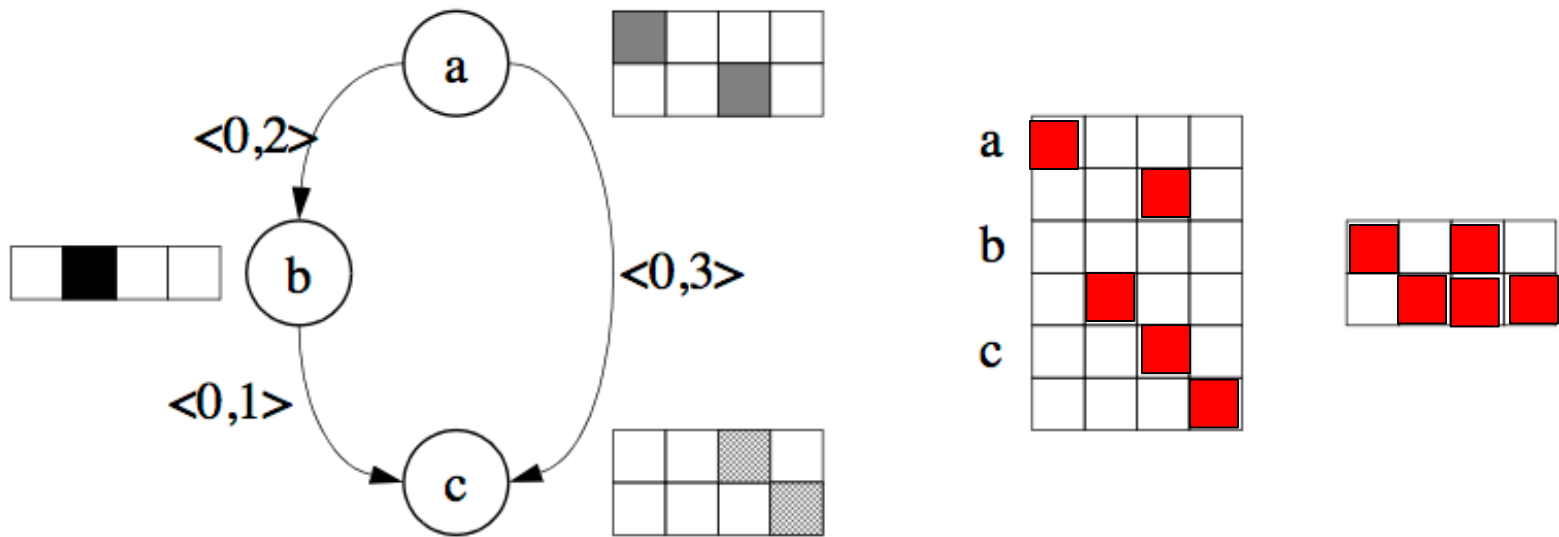
S(0) LD
 S(1) ADD
 S(2) LD
 S(3) ST

LD
 ADD
 ST

LD
 ADD
 ST

Label edges with $\langle \delta, d \rangle$: δ = iteration difference, d = delay

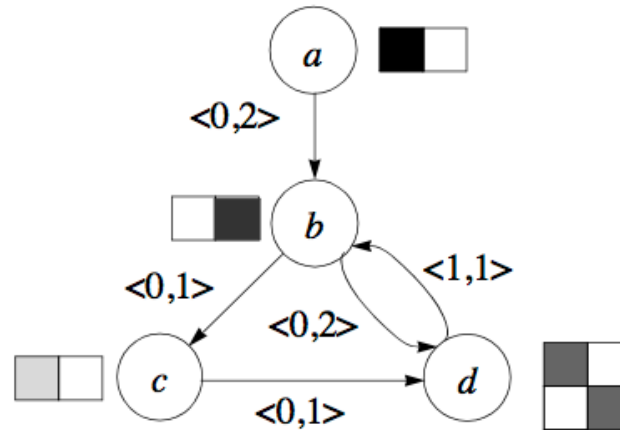
Example: An Acyclic Graph inside a loop



Algorithm: Software Pipelining Acyclic Dependence Graphs

- **Find lower bound of initiation interval: T_0**
 - based on resource constraints
- **For $T = T_0, T_0+1, \dots$ until all nodes are scheduled**
 - For each node n in **topological order**
 - $s_0 =$ earliest n can be scheduled
 - for each $s = s_0, s_0 + 1, \dots, s_0 + T - 1$
 - if `NodeScheduled(n, s)` break;
 - if n cannot be scheduled break;
- **NodeScheduled(n, s)**
 - Check resources of n at s in modulo resource reservation table
- Can always meet the lower bound if:
 - every operation **uses only 1 resource**, and
 - **no cyclic dependences** in the loop

Cyclic Graphs



- No such thing as “topological order”
- $b \rightarrow c; c \rightarrow b$

$$\delta \times T + S(n_2) - S(n_1) \geq d$$

$$S(c) - S(b) \geq 1$$

$$T + S(b) - S(c) \geq 2$$

- Scheduling b constrains c , and vice versa

$$S(b) + 1 \leq S(c) \leq S(b) - 2 + T$$

$$S(c) - T + 2 \leq S(b) \leq S(c) - 1$$

See [ALSU 10.5.8] for Software Pipelining scheduling algorithm for cyclic dependence graphs

A Closer Look at Register Allocation for Software Pipelining

Software-pipelined code:

```

1. LD
2. LD
3. MUL    LD
4.        LD
5.        MUL    LD
6. ADD    LD
L: 7.        MUL    LD
8. ST    ADD    LD    BL L
9.        MUL    LD
10.       ST    ADD    LD
11.       MUL
12.       ST    ADD
13.
14.       ST    ADD
    
```

```

1. LD  R5,0(R1++)
2. LD  R6,0(R2++)
3. MUL R7,R5,R6
4.
5.
6. ADD R8,R7,R4
7.
8. ST  0(R3++),R8
    
```

What is the problem w.r.t. R7?

R7 is clobbered before use

Solution: Modulo Variable Expansion

```
1. LD R5,0(R1++)
2. LD R6,0(R2++)
3. LD R5,0(R1++) MUL R7,R5,R6
4. LD R6,0(R2++)
5. LD R5,0(R1++) MUL R9,R5,R6
6. LD R6,0(R2++) ADD R8,R7,R4
L: 7. LD R5,0(R1++) MUL R7,R5,R6
   8. LD R6,0(R2++) ADD R8,R9,R4 ST 0(R3++),R8
   9. LD R5,0(R1++) MUL R9,R5,R6
  10. LD R6,0(R2++) ADD R8,R7,R4 ST 0(R3++),R8 BL L
  11. MUL R7,R5,R6
  12. ADD R8,R9,R4 ST 0(R3++),R8
  13.
  14. ADD R8,R7,R4 ST 0(R3++),R8
  15.
  16. ST 0(R3++),R8
```

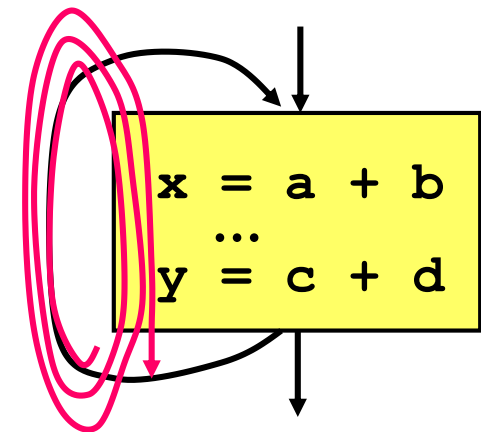
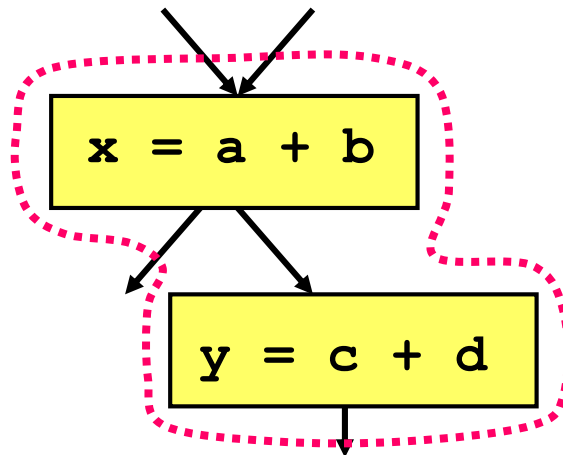
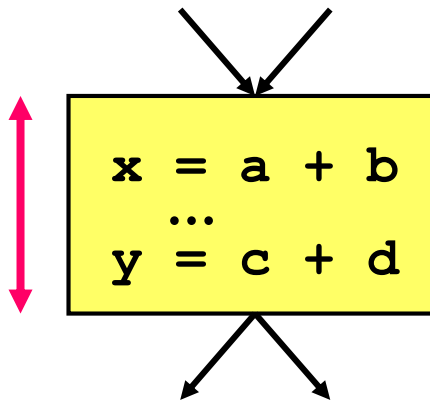

Algorithm: Software Pipelining with Modulo Variable Expansion

- **Normally, every iteration uses the same set of registers**
 - introduces **artificial anti-dependences** for software pipelining
- **Modulo variable expansion algorithm**
 - schedule each iteration ignoring artificial constraints on registers
 - calculate **life times of registers**
 - **degree of unrolling** = $\max_r (\text{lifetime}_r / T)$
 - **unroll** the steady state of software pipelined loop to **use different registers**
- **Code generation**
 - generate one pipelined loop with only one exit (at beginning of steady state)
 - generate one unpipelined loop to handle the rest
 - code generation is the messiest part of the algorithm!

Software Pipelining Summary

- **Numerical Code**

- Software pipelining is useful for machines with a lot of pipelining and instruction level parallelism
- Compact code
- Limits to parallelism: dependences, critical resource



List Scheduling:

- *within* a basic block

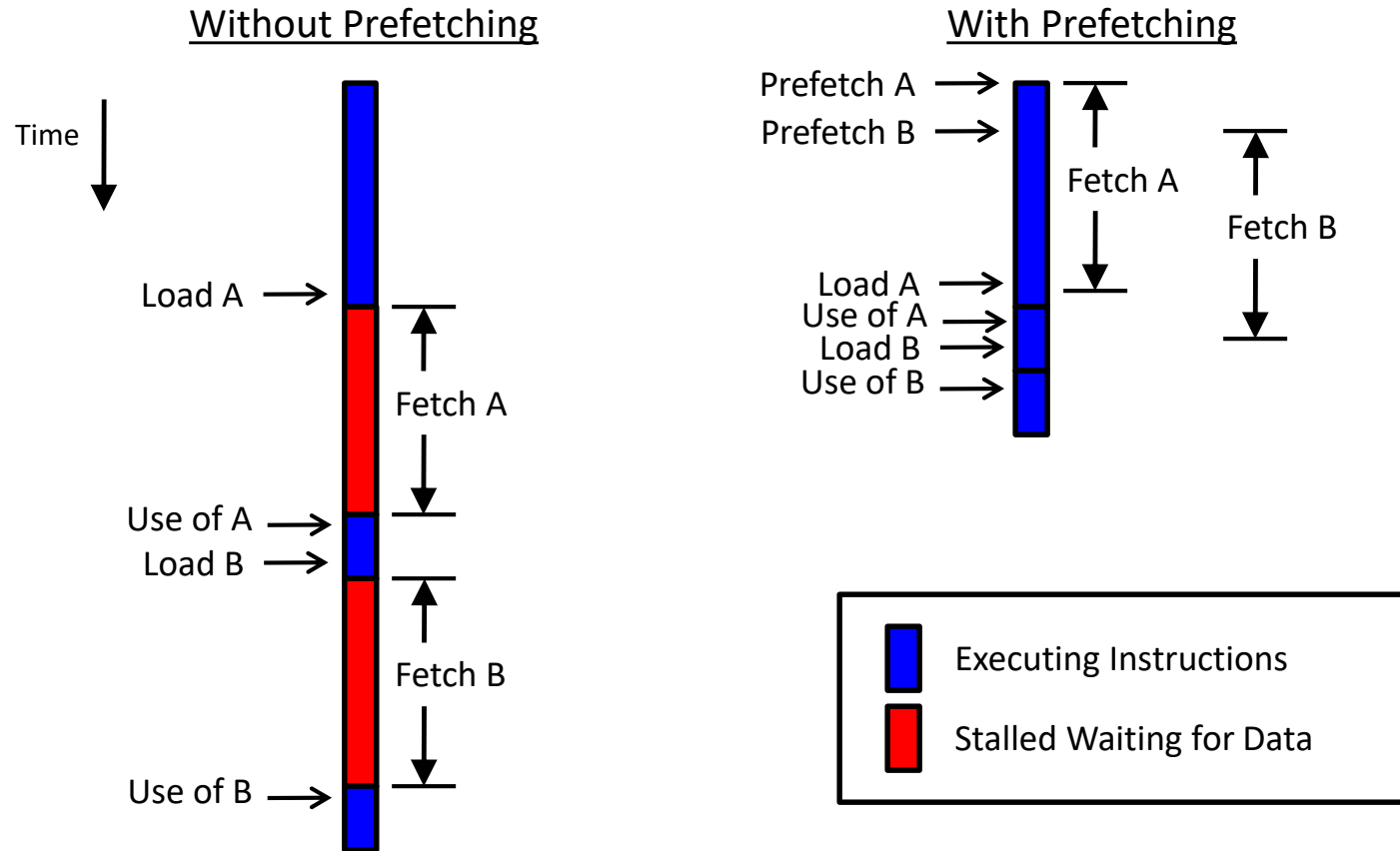
Global Scheduling:

- *across* basic blocks

Software Pipelining:

- *across* loop iterations

II. Software Prefetching



- **overlap memory accesses** with computation and other accesses
- used to **tolerate latency**

Types of Prefetching

Cache Blocks:

- + : no instruction overhead
- : best only for unit-stride accesses

Nonblocking Loads:

- + : no added instructions
- : limited ability to move back before use

Hardware-Controlled Prefetching:

- + : no instruction overhead
- : limited to constant-strides and by branch prediction

today

Software-Controlled Prefetching:

- + : minimal hardware support and broader coverage
- : software sophistication and overhead

Software Prefetching: Research Goals

- Domain of Applicability
- Performance Improvement
 - maximize benefit
 - minimize overhead

Prefetching Concepts

possible only if addresses can be determined ahead of time

coverage factor = fraction of misses that are prefetched

unnecessary if data is already in the cache

effective if data is in the cache when later referenced

Analysis: what to prefetch

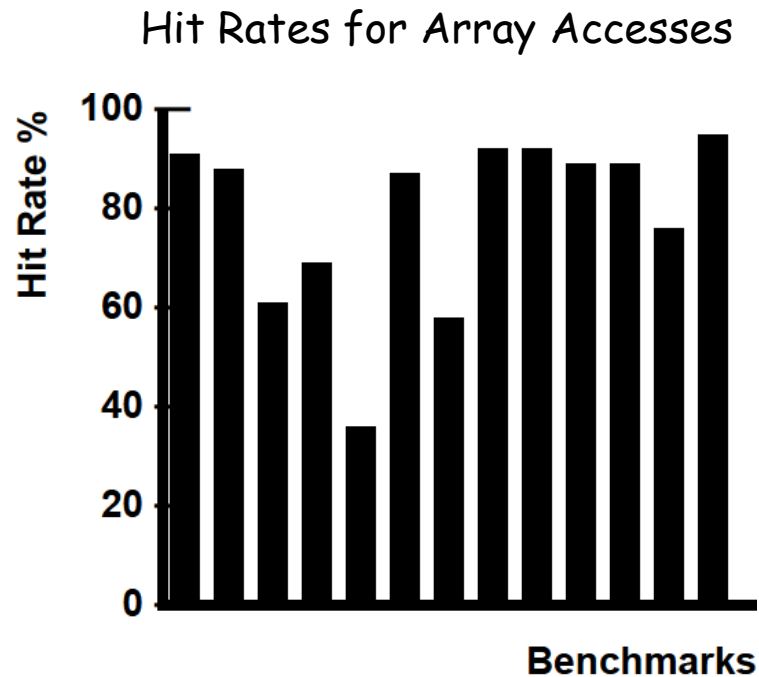
- maximize coverage factor
- minimize unnecessary prefetches

Scheduling: when/how to schedule prefetches

- maximize effectiveness
- minimize overhead per prefetch

Reducing Prefetching Overhead

- instructions to issue prefetches
- extra demands on memory system



- important to minimize unnecessary prefetches

Compiler Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

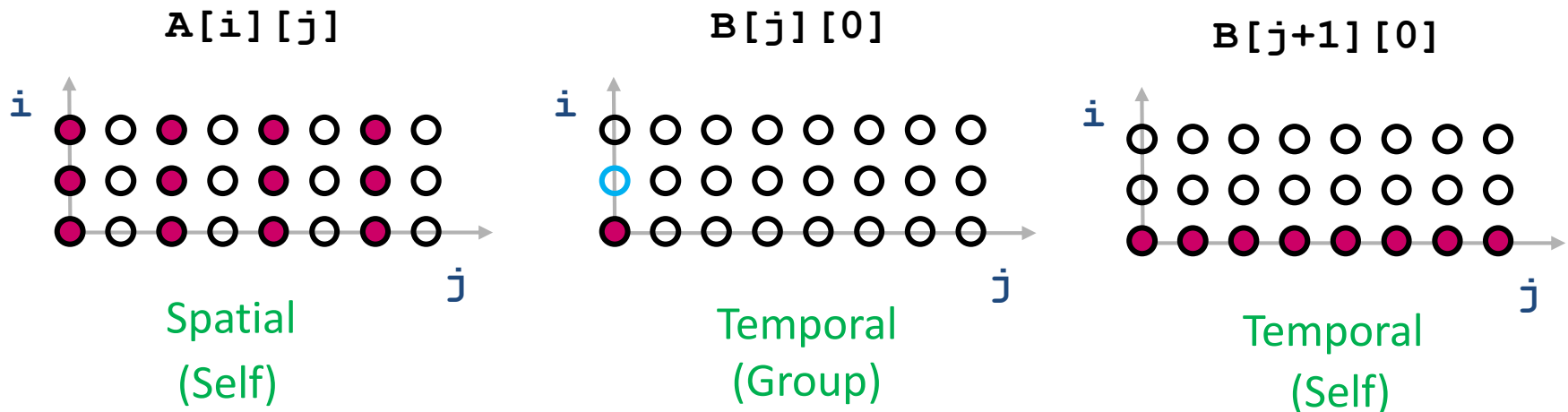
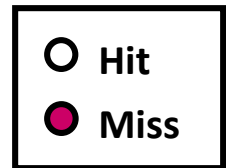
Recall: Types of Data Reuse/Locality

```
double A[3][N], B[N][3];
```

```
for i = 0 to 2
```

```
  for j = 0 to N-2
```

```
    A[i][j] = B[j][0] + B[j+1][0];
```



(assume row-major, 2 elements per cache line, N small)

Prefetch Predicate

Locality Type	Miss Instance	Predicate on Iteration Space
None	Every Iteration	True
Temporal	First Iteration	$i = 0$
Spatial	Every L iterations (L elements/cache line)	$(i \bmod L) = 0$

Example:

```

for i = 0 to 2
  for j = 0 to N-2
    A[i][j] = B[j][0] + B[j+1][0];
  
```

Reference	Locality	Predicate on Iteration Space
$A[i][j]$	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{none} \\ \text{spatial} \end{bmatrix}$	$(j \bmod L) = 0$
$B[j+1][0]$	$\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} \text{temporal} \\ \text{none} \end{bmatrix}$	$i = 0$

Compiler Algorithm

Analysis: what to prefetch

- Locality Analysis

Scheduling: when/how to issue prefetches

- Loop Splitting
- Software Pipelining

Loop Splitting

- Decompose loops to isolate cache miss instances
 - cheaper than inserting IF(Prefetch Predicate) statements

Locality Type	Predicate	Loop Transformation
None	True	None
Temporal	$i = 0$	Peel loop i
Spatial	$(i \bmod L) = 0$	Unroll loop i by L

(L elements/cache line)

Loop peeling: split any problematic first (or last) few iterations from the loop & perform them outside of the loop body

- Apply transformations recursively for nested loops
- Suppress transformations when loops become too large (avoid code explosion)

III. Prefetching via Software Pipelining

$$\text{Iterations Ahead} = \left\lceil \frac{l}{s} \right\rceil$$

where l = memory latency, s = shortest path through loop body

Original Loop

```
for (i = 0; i < 100; i++)  
  a[i] = 0;
```

Are there any
wasted prefetches?

Yes: $(L-1)/L$
are wasted

Software Pipelined Loop

(6 iterations ahead)

```
for (i = 0; i < 6; i++)           // Prologue  
  prefetch(&a[i]);  
  
for (i = 0; i < 94; i++) {       // Steady State  
  prefetch(&a[i+6]);  
  a[i] = 0;  
}  
  
for (i = 94; i < 100; i++)      // Epilogue  
  a[i] = 0;
```

Prefetching via Software Pipelining

$$\text{Iterations Ahead} = \left\lceil \frac{l}{s} \right\rceil$$

where l = memory latency, s = shortest path through loop body

Original Loop

```
for (i = 0; i < 100; i++)  
  a[i] = 0;
```

(2 elements/cache line)

Software Pipelined Loop

(6 iterations ahead)

```
for (i = 0; i < 6; i += 2) // Prologue  
  prefetch(&a[i]);  
  
for (i = 0; i < 94; i += 2) { // Steady State  
  prefetch(&a[i+6]);  
  a[i] = 0;  
  a[i+1] = 0;  
}  
  
for (i = 94; i < 100; i++) // Epilogue  
  a[i] = 0;
```

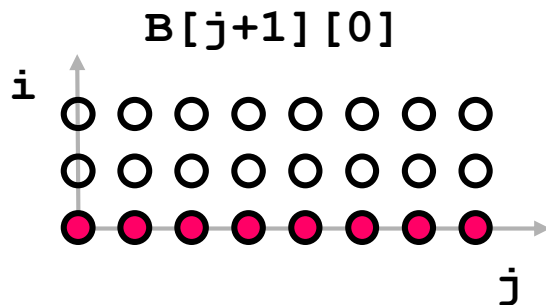
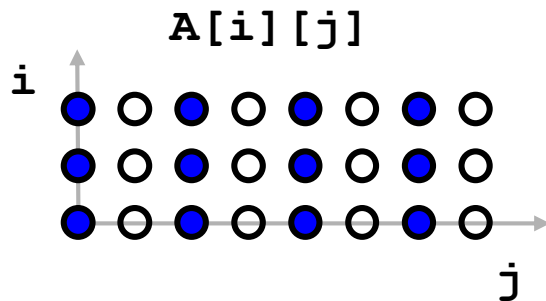
Example Code with Prefetching

Original Code

```
for (i = 0; i < 3; i++)
  for (j = 0; j < 100; j++)
    A[i][j] = B[j][0] + B[j+1][0];
```

○ Cache Hit

●● Cache Miss



```

prefetch(&B[0][0]);
for (j = 0; j < 6; j += 2) {
  prefetch(&B[j+1][0]);
  prefetch(&B[j+2][0]);
  prefetch(&A[0][j]);
}
for (j = 0; j < 94; j += 2) {
  prefetch(&B[j+7][0]);
  prefetch(&B[j+8][0]);
  prefetch(&A[0][j+6]);
  A[0][j] = B[j][0]+B[j+1][0];
  A[0][j+1] = B[j+1][0]+B[j+2][0];
}
for (j = 94; j < 100; j += 2) {
  A[0][j] = B[j][0]+B[j+1][0];
  A[0][j+1] = B[j+1][0]+B[j+2][0];
}

for (i = 1; i < 3; i++) {
  for (j = 0; j < 6; j += 2)
    prefetch(&A[i][j]);
  for (j = 0; j < 94; j += 2) {
    prefetch(&A[i][j+6]);
    A[i][j] = B[j][0] + B[j+1][0];
    A[i][j+1] = B[j+1][0] + B[j+2][0];
  }
  for (j = 94; j < 100; j += 2) {
    A[i][j] = B[j][0] + B[j+1][0];
    A[i][j+1] = B[j+1][0] + B[j+2][0];
  }
}
    
```

i = 0

i > 0

Prefetching Indirections

```
for (i = 0; i<100; i++)  
    sum += A[index[i]];
```

Analysis: what to prefetch

- both dense and **indirect** references
- difficult to predict whether indirections hit or miss

Scheduling: when/how to issue prefetches

- **modification of software pipelining algorithm**

Software Pipelining for Indirections

Original Loop

```
for (i = 0; i<100; i++)  
    sum += A[index[i]];
```

Software Pipelined Loop (5 iterations ahead)

```
for (i = 0; i<5; i++)           // Prolog 1  
    prefetch(&index[i]);  
  
for (i = 0; i<5; i++) {        // Prolog 2  
    prefetch(&index[i+5]);  
    prefetch(&A[index[i]]);  
}  
  
for (i = 0; i<90; i++) {      // Steady State  
    prefetch(&index[i+10]);  
    prefetch(&A[index[i+5]]);  
    sum += A[index[i]];  
}  
  
for (i = 90; i<95; i++) {    // Epilogue 1  
    prefetch(&A[index[i+5]]);  
    sum += A[index[i]];  
}  
  
for (i = 95; i<100; i++)     // Epilogue 2  
    sum += A[index[i]];
```

Today's Class: Software Pipelining & Prefetching

- I. Software Pipelining
- II. Software Prefetching (of Arrays)
- III. Prefetching via Software Pipelining

Wednesday's Class

- Locality Analysis & Prefetching
 - ALSU 11.5