Data Prefetch and Software Pipelining

Agenda

- *Data Prefetch*
- Software Pipelining

Why Data Prefetching

- Increasing Processor – Memory "distance"
- Caches do work !!! … IF …
  - Data set cache-able, accesses local (in space/time)
- Else ? …

Data Prefetching

- What is it ?
  - Request for a future data need is initiated
  - Useful execution continues during access
  - Data moves from slow/far memory to fast/near cache
  - Data ready in cache when needed (load/store)

Data Prefetching

- When can it be used ?
  - Future data needs are (somewhat) predictable
- How is it implemented ?
  - in hardware: history based prediction of future access
  - in software: compiler inserted prefetch instructions

Software Data Prefetching

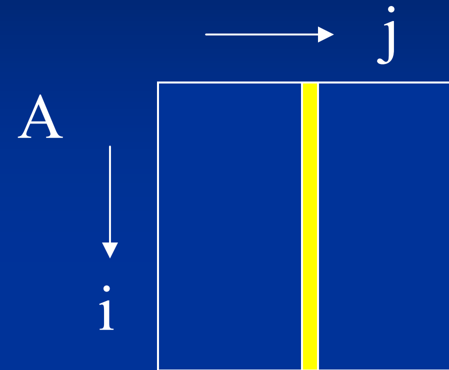
- Compiler scheduled prefetches
- Moves entire cache lines (not just datum)
  - Spatial locality assumed – often the case
- Typically a non-faulting access
  - Compiler free to speculate prefetch address
- Hardware not obligated to obey
  - A performance enhancement, no functional impact
  - Loads/store may be preferentially treated

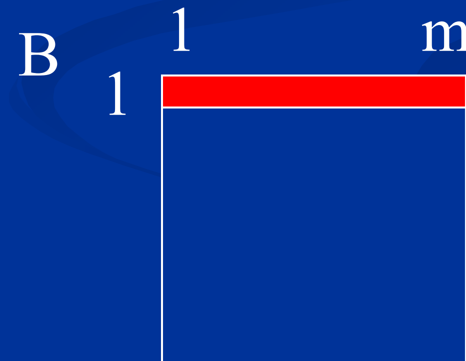
Software Data Prefetching Use

- Mostly in Scientific codes
  - Vectorizable loops accessing arrays deterministically
    - Data access pattern is predictable
    - Prefetch scheduling easy (far in time, near in code)
  - Large working data sets consumed
    - Even large caches unable to capture access locality
- Sometimes in Integer codes
  - Loops with pointer de-references

Selective Data Prefetch

```
do j = 1, n
  do i = 1, m
     A(i,j) = B(1,i) + B(1,i+1)
  enddo
enddo
```





- E.g. A(i,j) has spatial locality, therefore only one prefetch is required for every cache line.

Formal Definitions

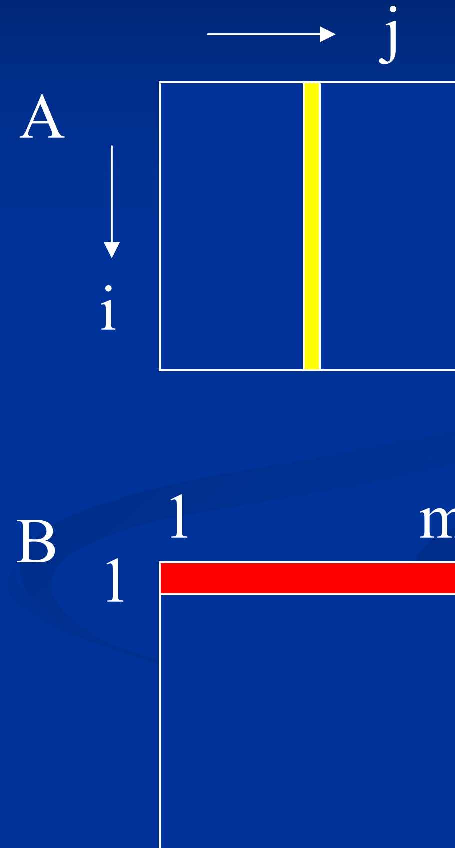
- *Temporal locality* occurs when a given reference reuses exactly the same data location
- *Spatial locality* occurs when a given reference accesses different data locations that fall within the same cache line
- *Group locality* occurs when different references access the same cache line

Prefetch Predicates

- If an access has spatial locality, only the first access to the same cache line will incur a miss.
- For temporal locality, only the first access will incur a cache miss
- If an access has group locality, only the leading reference incurs cache miss.
- If an access has no locality, it will miss in every iteration.

Example Code with Prefetches

```
do j = 1, n
  do i = 1, m
    A(i,j) = B(1,i) + B(1,i+1)
    if (iand(i,7) == 0)
      prefetch (A(i+k,j))
    if (j == 1)
      prefetch (B(1,i+t))
  enddo
enddo
```



Assumed CLS = 64 bytes and data size = 8 bytes

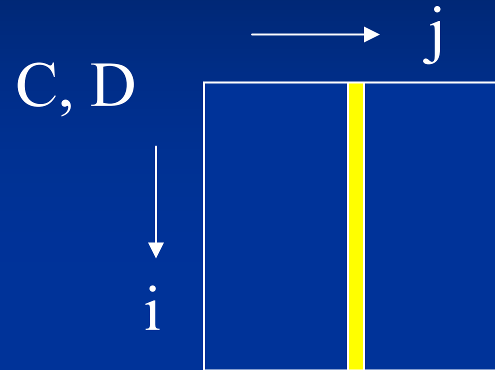
k and t are prefetch distance values

Spreading of Prefetches

- If there is more than one reference that has spatial locality within the same loop nest, spread these prefetches across the 8-iteration window
- Reduces the stress on the memory subsystem by minimizing the number of outstanding prefetches

Example Code with Spreading

```
do j = 1, n
  do i = 1, m
     C(i,j) = D(i-1,j) + D(i+1,j)
     if (iand(i,7) == 0)
       prefetch (C(i+k,j))
      if (iand(i,7) == 1)
       prefetch (D(i+k+1,j))
    enddo
enddo
```



Assumed CLS = 64 bytes and data size = 8 bytes

k is the prefetch distance value

Prefetch Strategy - Conditional

Example loop

```
L:
  Load A(I)
  Load B(I)
  ...
  I = I + 1
  Br L, if I<n
```

- ↓Code for condition generation
- ↓Prefetches occupy issue slots

Conditional Prefetching

```
L:
  Load A(I)
  Load B(I)
  Cmp pA=(I mod 8 == 0)
  if(pA) prefetch
  A(I+X)
  Cmp pB=(I mod 8 == 1)
  If(pB) prefetch
  B(I+X)
  ...
  I = I + 1
  Br L, if I<n
```

Prefetch Strategy - Unroll

Example loop

```
L:
  Load A(I)
  Load B(I)
  ...
  I = I + 1
  Br L, if I<n
```

Unrolled

```
Unr_Loop:
  prefetch A(I+X)
  load A(I)
  load B(I)
  ...
  prefetch B(I+X)
  load A(I+1)
  load B(I+1)
  ...
  prefetch C(I+X)
  load A(I+2)
  load B(I+2)
  ...
  prefetch D(I+X)
  load A(I+3)
  load B(I+3)
  ...
  prefetch E(I+X)
  load A(I+4)
  load B(I+4)

  ...
  load A(I+5)
  load B(I+5)

  ...
  load A(I+6)
  load B(I+6)

  ...
  load A(I+7)
  load B(I+7)
  ...
  I = I + 8
  Br Unr_Loop, if
  I<n
```

- ↓**Code bloat (>8X)**
- ↓**Remainder loop**

Software Data Prefetching Cost

- Requires memory instruction resources
  - A prefetch instruction for each access stream
- Issues every iteration, but needed less often
  - If branched around, inefficient execution results
  - If conditionally executed, more instruction overhead results
  - If loop is unrolled, code bloat results

Software Data Prefetching Cost

- Redundant prefetches get in the way
  - Resources consumed until prefetches discarded!
- Non redundant need careful scheduling
  - Resources overwhelmed when many issued & miss

Desirable Characteristics

- Uses minimal instruction resources
  - One prefetch instruction for multiple streams
- Minimizes redundant prefetches
  - No code bloat, no prefetch branches
- Issues prefetches spaced in time
  - Machine resources utilized evenly
- Solution: rotating register prefetch if there is HW support.

Rotating Registers

- Register rotation provides an automatic renaming mechanism.
- Instructions contain a "virtual" register number

	32 33 34 35 36 37 38 39
Iteration 1	r32r33r34r35r36r37r38r39
Iteration 2	r33r34r35r36r37r38r39r32
Iteration 3	r34r35r36r37r38r39r32r33

Rotating Reg Prefetch Illustrated

Example loop

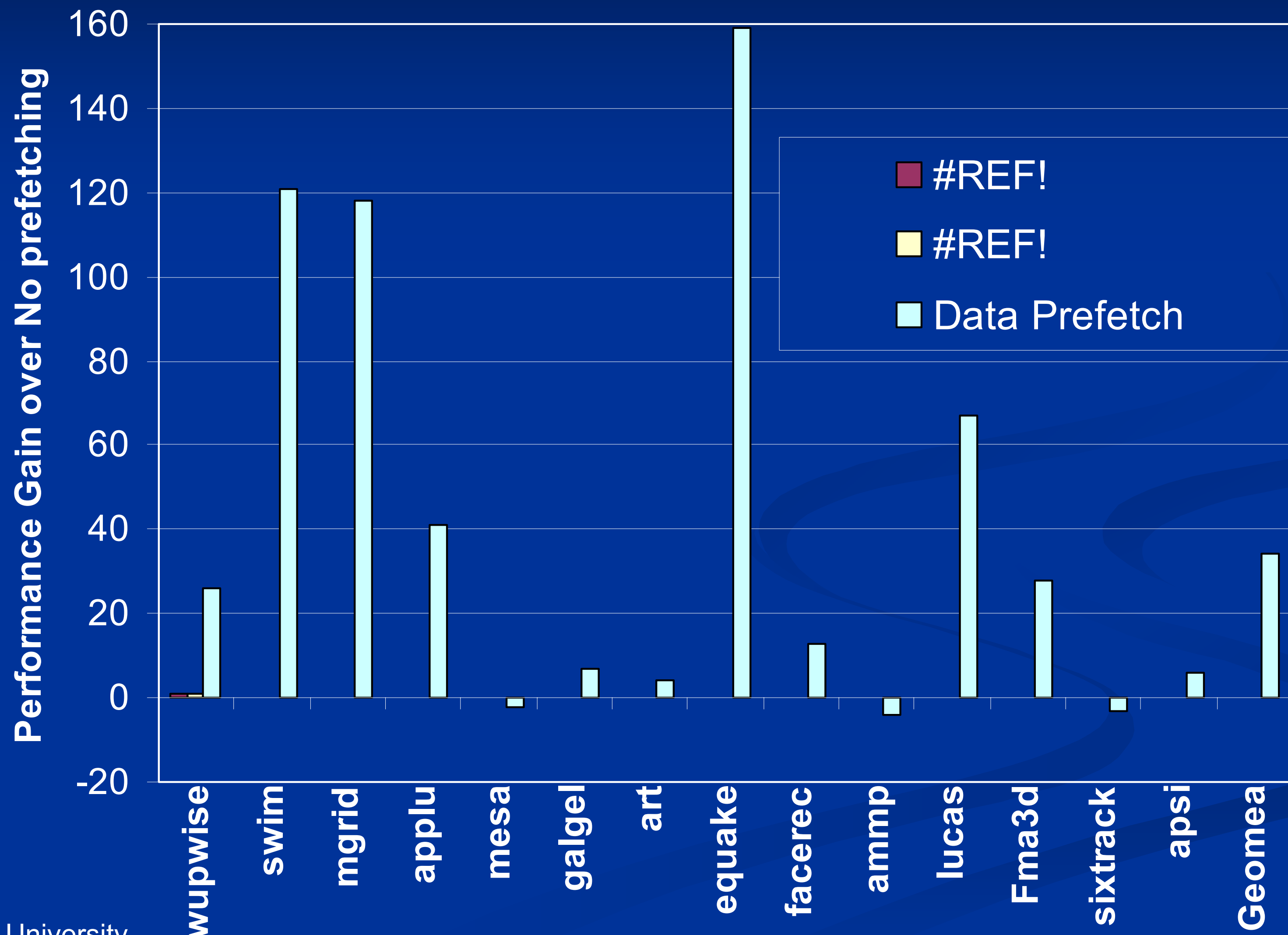
```
Orig_loop:
   Load A(I)
   Load B(I)
   Load C(I)
   Load D(I)
   Load E(I)
   ...
   I = I + 1
   Br Orig_loop,
   if I<n
```

- ↑Single prefetch inst
- ↑No loop unrolling
- ↑At most 1 miss/iter

Rotating Register Prefetching

```
r33 = address of E(1+X)
r34 = address of D(1+X)
r35 = address of C(1+X)
r36 = address of B(1+X)
r37 = address of A(1+X)
Method1Loop:
prefetch [r37]
r32 = r37 + INCR
. . .
load A(I)
load B(I)
load C(I)
load D(I)
load E(I)
. . .
I = I + 1
Br Method1Loop, if I<n
```

Measurements – SPECfp2000



Agenda

- Data Prefetch
- *Software Pipelining*

Software Pipelining

- Obtain parallelism by executing iterations of a loop in an overlapping way.
- We'll focus on simplest case: the *do-all* loop, where iterations are independent.
- Goal: Initiate iterations as frequently as possible.
- Limitation: Use same schedule and delay for each iteration.

Machine Model

- Timing parameters: LD = 2, others = 1 clock.
- Machine can execute one LD or ST and one arithmetic operation (including branch) at any one clock.
  - I.e., we're back to one ALU resource and one MEM resource.

Example

```
for (i=0; i<N; i++)
    B[i] = A[i];
```

- r9 holds 4N; r8 holds 4*i.

```
L:  LD r1, a(r8)
    nop
    ST b(r8), r1
    ADD r8, r8, #4
    BLT r8, r9, L
```

Notice: data dependences force this schedule. No parallelism is possible.

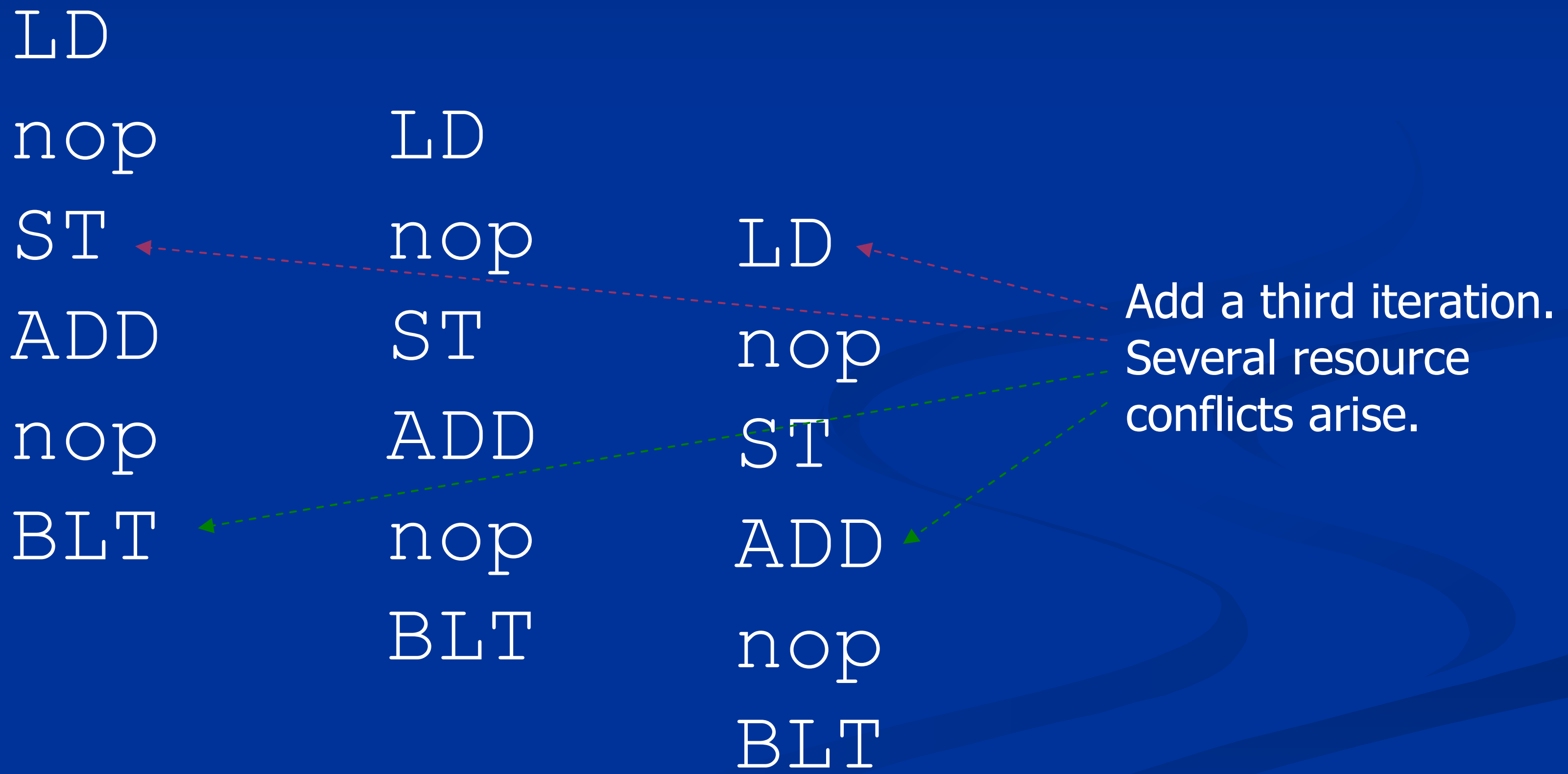
Let's Run 2 Iterations in Parallel

- Focus on operations; worry about registers later.

```
LD
nop     LD
ST      nop
ADD     ST
BLT     ADD
        BLT
```

Oops --- violates ALU resource constraint.

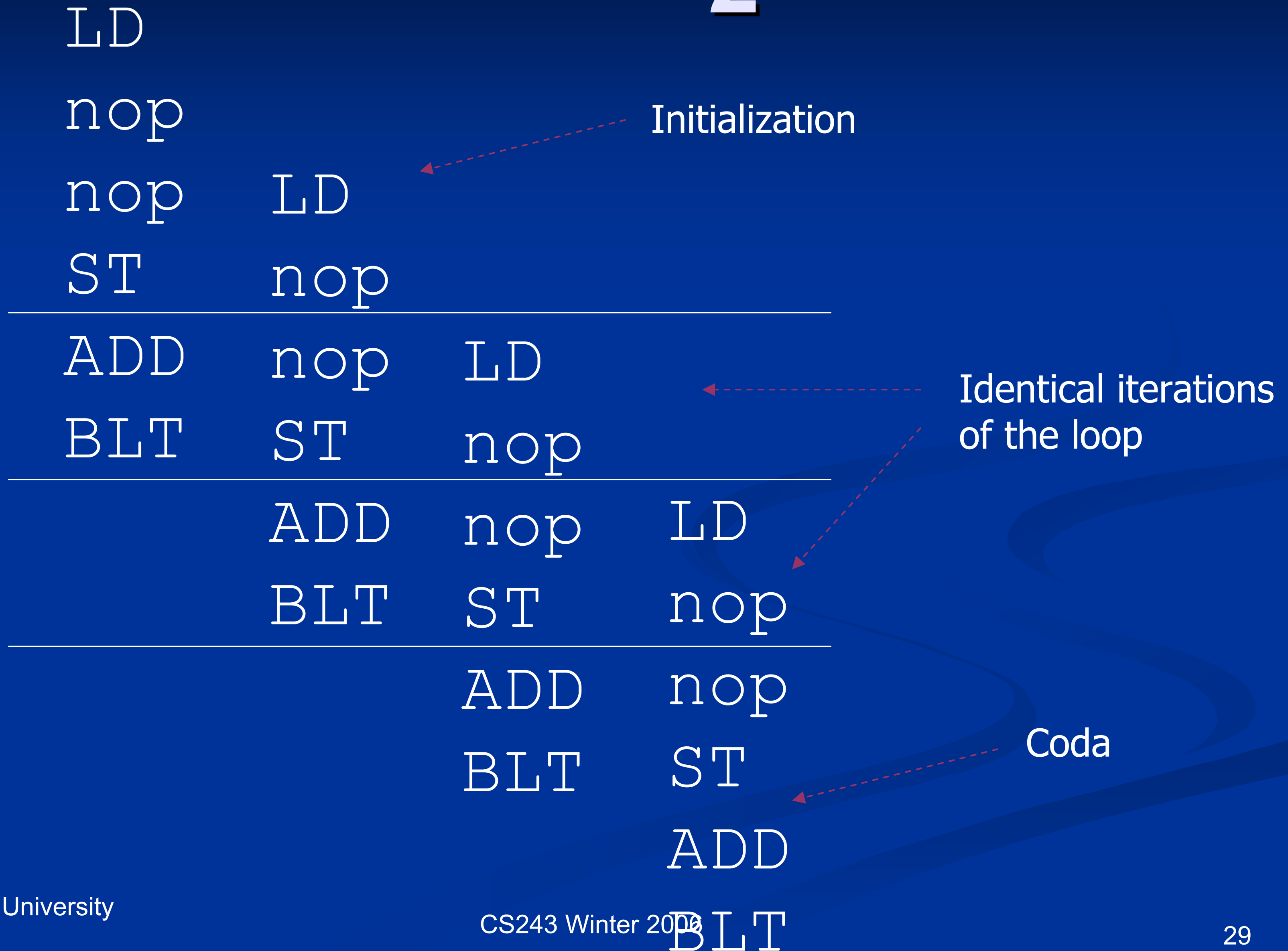
Introduce a NOP



Is It Possible to Have an Iteration Start at Every Clock?

- Hint: No.
- Why?
- An iteration injects 2 MEM and 2 ALU resource requirements.
  - If injected every clock, the machine cannot possibly satisfy all requests.
- Minimum delay = 2.

A Schedule With Delay 2



Assigning Registers

- We don’t need an infinite number of registers.
- We can reuse registers for iterations that do not overlap in time.
- But we can’t just use the same old registers for every iteration.

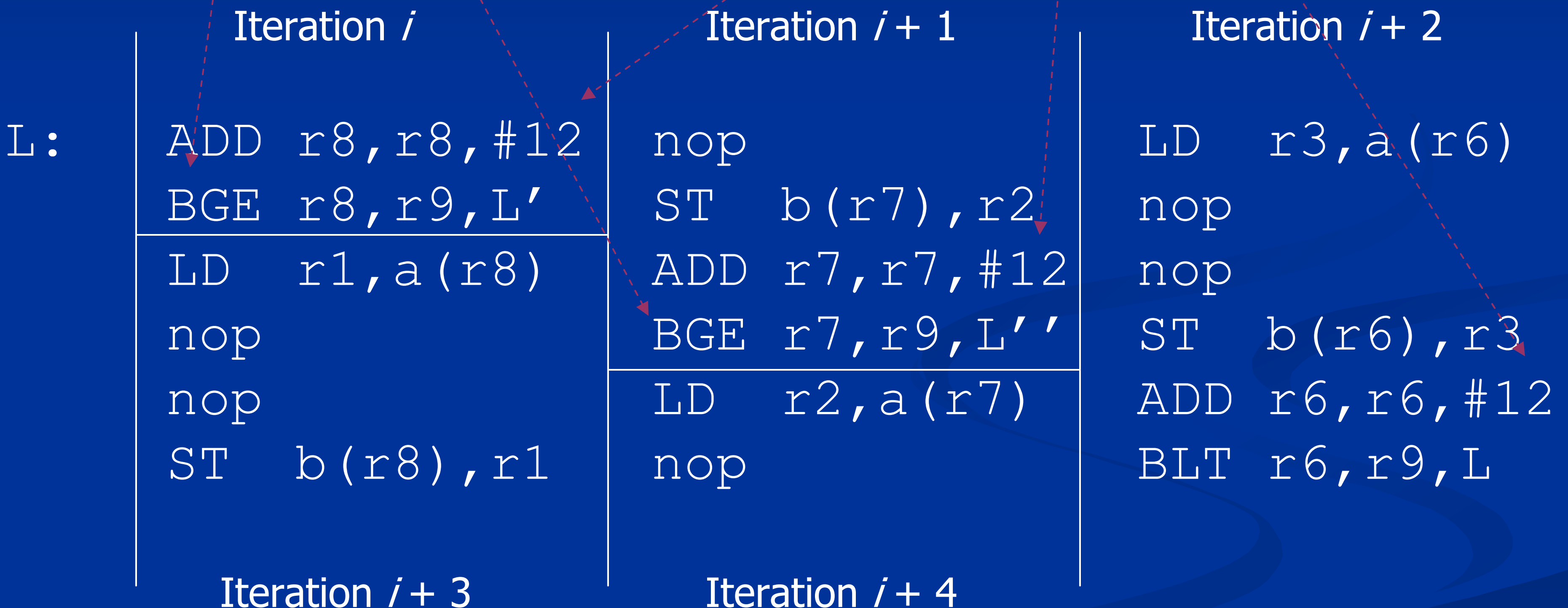
Assigning Registers --- (2)

- The inner loop may have to involve more than one copy of the smallest repeating pattern.
  - Enough so that registers may be reused at each iteration of the expanded inner loop.
- Our example: 3 iterations coexist, so we need 3 sets of registers and 3 copies of the pattern.

Example: Assigning Registers

- Our original loop used registers:
  - r9 to hold a constant 4N.
  - r8 to count iterations and index the arrays.
  - r1 to copy a[i] into b[i].
- The expanded loop needs:
  - r9 holds 12N.
  - r6, r7, r8 to count iterations and index.
  - r1, r2, r3 to copy certain array elements.

The Loop Body

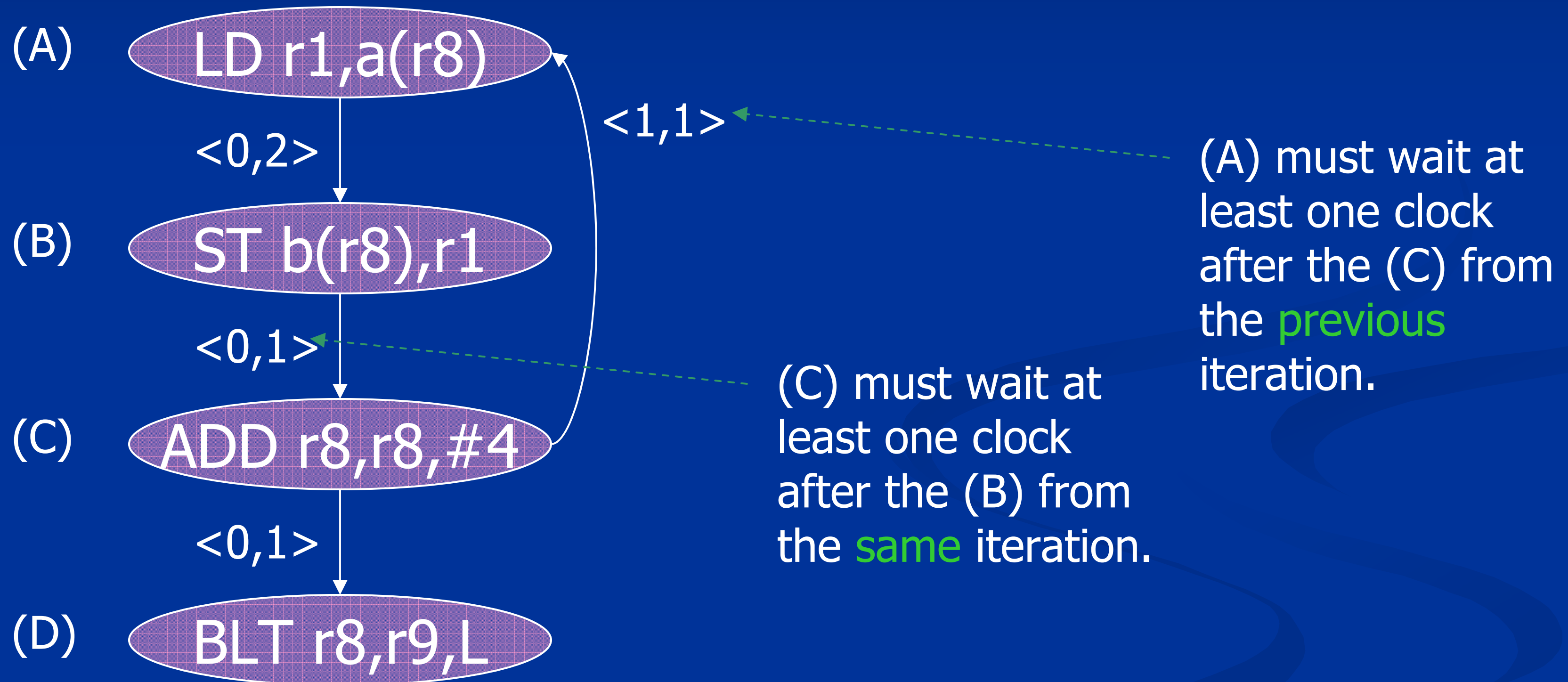


L' and L" are places for appropriate codas.

Cyclic Data-Dependence Graphs

- We assumed that data at an iteration depends only on data computed at the same iteration.
  - Not even true for our example.
    - r8 computed from its previous iteration.
    - But it doesn't matter in this example.
- Fixup: edge labels have two components: (iteration change, delay).

Example: Cyclic D-D Graph



Matrix of Delays

- Let T be the delay between the start times of one iteration and the next.
- Replace edge label <i,j> by delay j-iT.
- Compute, for each pair of nodes n and m the total delay along the longest acyclic path from n to m.
- Gives upper and lower bounds relating the times to schedule n and m.

Example: Delay Matrix

	A	B	C	D
A		2		
B			1	
C	1-T			1
D				

Edges

	A	B	C	D
A		2	3	4
B	2-T		1	2
C	1-T	3-T		1
D				

Acyclic Transitive Closure

Note: Implies $T \geq 4$ (because only one register used for loop-counting). If T=4, then A (LD) must be 2 clocks before B (ST). If T=5, A can be 2-3 clocks before B.

$S(B) \geq S(A)+2$

$S(A) \geq S(B)+2-T$

$S(B)-2 \geq S(A) \geq S(B)+2-T$

Iterative Modulo Scheduling

- Compute the lower bounds (*MII*) on the delay between the start times of one iteration and the next (*initiation interval, aka II*)
  - due to resources
  - due to recurrences
- Try to find a schedule for II = MII
- If no schedule can be found, try a larger II.

Summary

- References for compiler data prefetch:
  - Todd Mowry, Monica Lam, Anoop Gupta, “Design and evaluation of a compiler algorithm for prefetching”, in ASPLOS’92, http://citeseer.ist.psu.edu/mowry92design.html.
  - Gautam Doshi, Rakesh Krishnaiyer, Kalyan Muthukumar, “Optimizing Software Data Prefetches with Rotating Registers”, in PACT’01, http://citeseer.ist.psu.edu/670603.html.