15-745 Lecture 5

Control flow analysis
Natural loops
Classical Loop Optimizations
Dependencies

Based on slides from Lee & Callahan

Loops are Key

- Loops are **extremely** important
  - the "90-10" rule
- Loop optimization involves
  - understanding control-flow structure
  - Understanding data-dependence information
  - sensitivity to side-effecting operations
  - extra care in some transformations such as register spilling

Common loop optimizations

- Hoisting of loop-invariant computations
  - pre-compute before entering the loop
- Elimination of induction variables
  - change p=i*w+b to p=b,p+=w, when w,b invariant
- Loop unrolling
  - to improve scheduling of the loop body

Scalar opts, DF analysis, Control flow analysis

- Software pipelining
  - To improve scheduling of the loop body
- Loop permutation
  - to improve cache memory performance

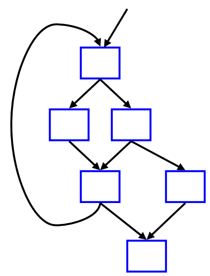
Requires understanding data dependencies

Finding Loops

- To optimize loops, we need to find them!
- Could use source language loop information in the abstract syntax tree…
- BUT:
  - There are multiple source loop constructs: for, while, do-while, even goto in C
  - Want IR to support different languages
  - Ideally, we want a single concept of a loop so all have same analysis, same optimizations
  - Solution: dismantle source-level constructs, then re-find loops from fundamentals

Finding Loops

- To optimize loops, we need to find them!
- Specifically:
  - loop-header node(s)
    - nodes in a loop that have immediate predecessors not in the loop
  - back edge(s)
    - control-flow edges to previously executed nodes
  - all nodes in the loop body



Control-flow analysis

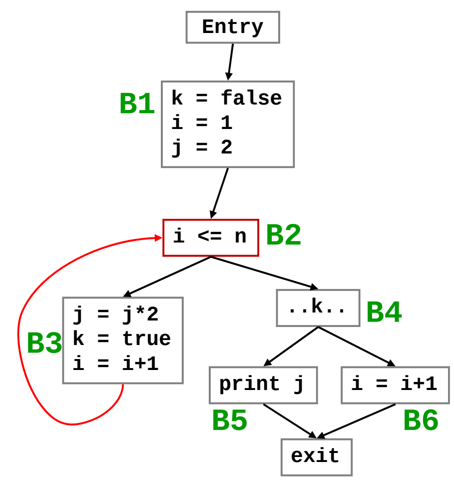
- Many languages have goto and other complex control, so loops can be hard to find in general
- Determining the control structure of a program is called control-flow analysis
- Based on the notion of dominators

Dominators

- a dom b
  - node a dominates b if every possible execution path from entry to b includes a
- a sdom b
  - a strictly dominates b if a dom b and a != b
- a idom b
  - a immediately dominates b if a sdom b, AND there is no c such that a sdom c and c sdom b

Back edges and loop headers

- A control-flow edge from node B3 to B2 is a back edge if
  B2 dom B3
- Furthermore, in that case node B2 is a loop header

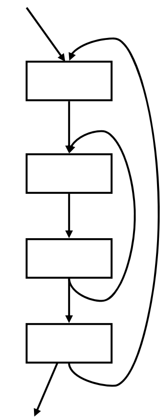


Natural loop

- Consider a back edge from node n to node h
- The natural loop of n→h is the set of nodes L such that for all x∈L:
  - h dom x and
  - there is a path from x to n not containing h

Examples

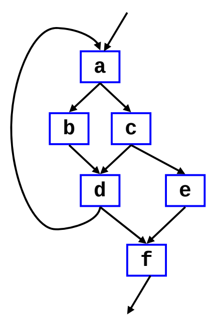
Simple example:



(often it's more complicated, since a FOR loop found in the source code might need an if/then guard)

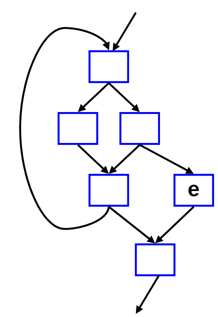
Examples

Try this:



Examples

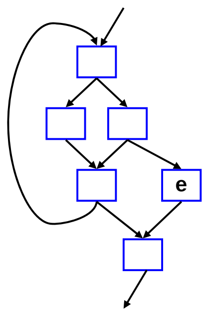
```
for (..) {
  if {
  …
  } else {
    …
    if (x) {
        e;
        break;
    )
  }
}
```



Examples

```
for (..) {
  if {
  …
  } else {
    …
    if (x) {
        e;
        break;
    )
  }
}
```

lexically, in loop, but not in natural loop

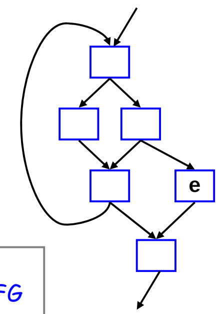


Examples

```
for (..) {
  if {
  …
  } else {
    …
    if (x) {
        e;
        break;
    )
  }
}
```

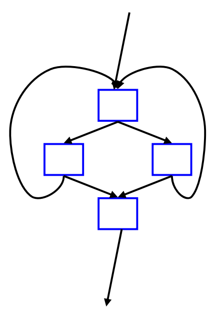
lexically, in loop, but not in natural loop

and another reason why CFG analysis is preferred over source/AST loops



Examples

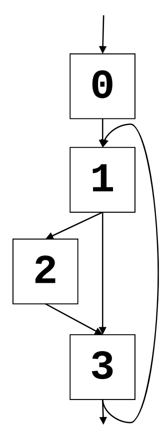
- Yes, it can happen in C



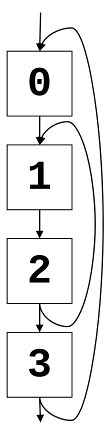
Natural Loops

What are the natural loops?

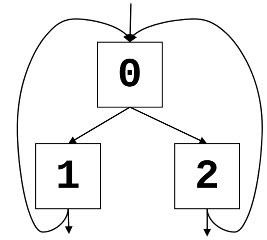
One loop per header..



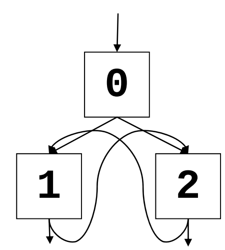
{1,2,3}



{1,2},{0,1,2,3}



{0,1,2}



{}

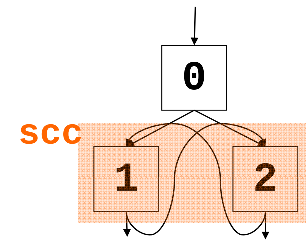
Nested Loops

- Unless two natural loops have the same header, they are either disjoint or nested within each other
- If A and B are loops (sets of blocks) with headers a and b such that $a \neq b$ and $b \in A$
  - $B \subset A$
  - loop B is *nested* within A
  - B is the *inner loop*
- Can compute the loop-nest tree

General Loops

- A more general looping structure is a *strongly connected component* of the control flow graph
  - subgraph $\langle N_{scc}, E_{scc} \rangle$ such that

    every block in N_{scc} is reachable from every other node using only edges in E_{scc}



Not very useful definition of a loop

Reducible Flow Graphs

There is a special class of flow graphs, called reducible flow graphs, for which several code-optimizations are especially easy to perform.

In reducible flow graphs loops are unambiguously defined and dominators can be efficiently computed.

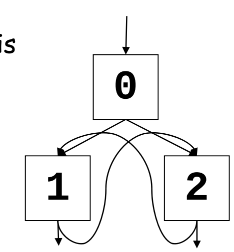
Reducible flow graphs

Definition: A flow graph G is reducible iff we can partition the edges into two disjoint groups, forward edges and back edges, with the following two properties.

1. The forward edges form an acyclic graph in which every node can be reached from the initial node of G.

2. The back edges consist only of edges whose heads dominate their tails.

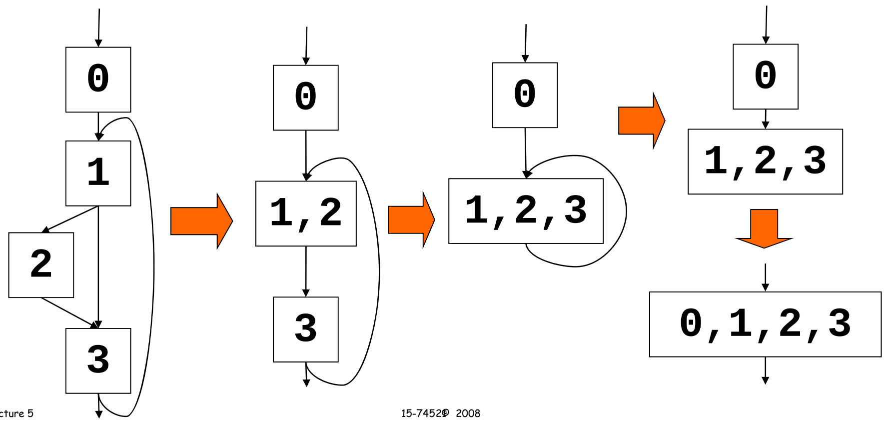
Why isn't this reducible?



This flow graph has no back edges. Thus, it would be reducible if the entire graph were acyclic, which is not the case.

Alternative definition

- Definition: A flow graph G is reducible if we can repeatedly collapse (reduce) together blocks (*x*,y) where x is the only predecessor of y (ignoring self loops) until we are left with a single node



Properties of Reducible Flow Graphs

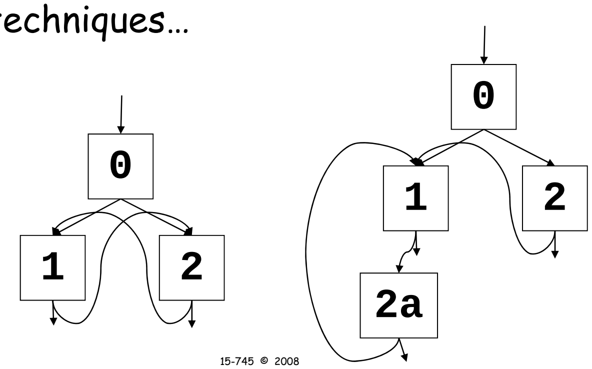
- In a reducible flow graph, **all** loops are natural loops
- Can use DFS to find loops
- Many analyses are more efficient
  - polynomial versus exponential

Good News

- Most flow graphs are reducible
- Languages can prohibit irreducibility
  - goto free C
  - Java
- Programmers usually don't use such constructs even if they're available
  - >90% of old Fortran code reducible

Dealing with Irreducibility

- Don't
- Can split nodes and duplicate code to get reducible graph
  - possible exponential blowup
- Other techniques...



Loop optimizations: Hoisting of loop-invariant computations

Loop-invariant computations

- A definition

  t = x op y

  in a loop is (conservatively) loop-invariant if
  - x and y are constants, or
  - all reaching definitions of x and y are outside the loop, or
  - only one definition reaches x (or y), and that definition is loop-invariant
    - so keep marking iteratively

Loop-invariant computations

- Be careful:

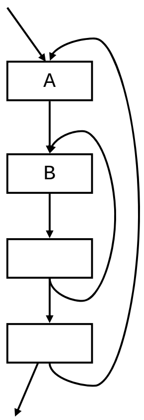
```
t = expr;
for () {
    s = t * 2;
    t = loop_invariant_expr;
    x = t + 2;
    …
}
```

- Even though t's two reaching expressions are each invariant, s is not invariant…

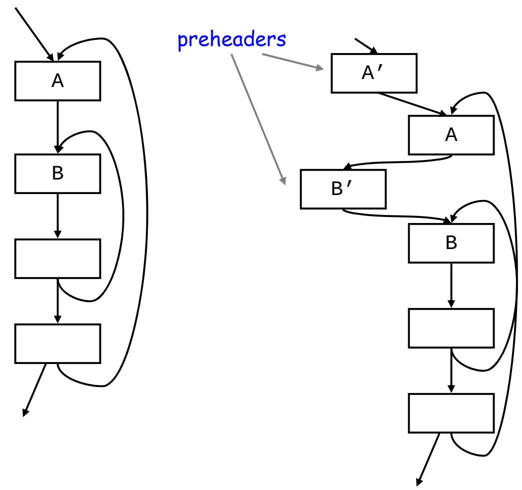
Hoisting

- In order to "hoist" a loop-invariant computation out of a loop, we need a place to put it
- We could copy it to all immediate predecessors (except along the back-edge) of the loop header...
- ...But we can avoid code duplication by inserting a new block, called the pre-header

Hoisting



Hoisting



Hoisting conditions

- For a loop-invariant definition

  d: t = x op y

- we can hoist d into the loop's pre-header only if
  1. d's block dominates all loop exits at which t is live-out, and
  2. d is only the only definition of t in the loop, and
  3. t is not live-out of the pre-header

We need to be careful...

- All hoisting conditions must be satisfied!

```
L0:
 t = 0
L1:
 i = i + 1
 t = a * b
 M[i] = t
 if i<N goto L1
L2:
 x = t
```

OK

```
L0:
 t = 0
L1:
 if i>=N goto L2
 i = i + 1
 t = a * b
 M[i] = t
 goto L1
L2:
 x = t
```

violates 1,3

```
L0:
 t = 0
L1:
 i = i + 1
 t = a * b
 M[i] = t
 t = 0
 M[j] = t
 if i<N goto L1
L2:
```

violates 2

We need to be careful...

- All hoisting conditions must be satisfied!

```
L0:
 t = 0
L1:
 i = i + 1
 t = a * b
 M[i] = t
 if i<N goto L1
L2:
 x = t
```

OK

```
L0:
 t = 0
L1:
 if i>=N goto L2
 i = i + 1
 t = a * b
 M[i] = t
 goto L1
L2:
 x = t
```

this def reaches

violates 1,3

```
L0:
 t = 0
L1:
 i = i + 1
 t = a * b
 M[i] = t
 t = 0
 M[j] = t
 if i<N goto L1
L2:
```

this def reaches

violates 2

Loop optimizations: Induction-variable Strength reduction

The basic idea of IVE

- Suppose we have a loop variable
  - i initially 0; each iteration i = i + 1
- and a variable that linearly depends on it:

  $x = i * c1 + c2$
- In such cases, we can try to
  - initialize $x = i_o * c1 + c2$ (execute once)
  - increment x by c1 each iteration

Simple Example of IVE

```
for i = 0 to n
    a[i] = 0
```

⇒

```
    i <- 0
H:
    if i >= n goto exit
    j <- i * 4
    k <- j + a
    M[k] <- 0
    i <- i + 1
    goto H
```

Clearly, j & k do not need to be computer anew each time since they are related to i and i changes linearly.

Simple Example of IVE

```
     i <- 0
H:
     if i >= n goto exit
     j <- i * 4
     k <- j + a
     M[k] <- 0
     i <- i + 1
     goto H
```

```
     i <- 0
     j' <- 0
     k' <- a
H:
     if i >= n goto exit
     j <- j'
     k <- k'
     M[k] <- 0
     i <- i + 1
     j' <- j' + 4
     k' <- k' + 4
     goto H
```

But, then we don't even need j (or j')

Simple Example of IVE

```
     i <- 0
     j' <- 0
     k' <- a
H:
     if i >= n goto exit
     j <- j'
     k <- k'
     M[k] <- 0
     i <- i + 1
     j' <- j' + 4
     k' <- k' + 4
     goto H
```

```
     i <- 0
     k' <- a
H:
     if i >= n goto exit
     k <- k'
     M[k] <- 0
     i <- i + 1
     k' <- k' + 4
     goto H
```

Do we need i?

Simple Example of IVE

Rewrite comparison

```
     i <- 0
     k' <- a
H:
     if i >= n goto exit
     k <- k'
     M[k] <- 0
     i <- i + 1
     k' <- k' + 4
     goto H
```

```
     i <- 0
     k' <- a
H:
     if k' >= a+(n*4) goto exi
     k <- k'
     M[k] <- 0
     k' <- k' + 4
     goto H
```

But, a+(n*4) is loop invariant

Simple Example of IVE

Invariant code motion on a+(n*4)

```
     i <- 0
     k' <- a
H:
     if k' >= a+(n*4)goto exit
     k <- k'
     M[k] <- 0
     k' <- k' + 4
     goto H
```

```
     k' <- a
     n' <- a + (n * 4)
H:
     if k' >= n' goto exit
     k <- k'
     M[k] <- 0
     k' <- k' + 4
     goto H
```

now, we do copy propagation and eliminate k

Simple Example of IVE

Copy propagation

```
     k' <- a
     n' <- a + (n * 4)
H:
     if k' >= n' goto exit
     k <- k'
     M[k] <- 0
     k' <- k' + 4
     goto H
```

⇒

```
     k' <- a
     n' <- a + (n * 4)
H:
     if k' >= n' goto exit
     M[k'] <- 0
     k' <- k' + 4
     goto H
```

Voila!

Simple Example of IVE

Compare original and result of IVE

```
   i <- 0
H:
   if i >= n goto exit
   j <- i * 4
   k <- j + a
   M[k] <- 0
   i <- i + 1
   goto H
```

```
   k' <- a
   n' <- a + (n * 4)
H:
   if k' >= n' goto exit
   M[k'] <- 0
   k' <- k' + 4
   goto H
```

Voila!

What we did

- identified induction variables (i,j,k)
- strength reduction (changed * into +)
- dead-code elimination (j <- j')
- useless-variable elimination (j' <- j' + 4)
  (This can also be done with ADCE)
- loop invariant identification & code-motion
- almost useless-variable elimination (i)
- copy propagation

Is it faster?

- On some hardware, adds are much faster than multiplies
- Furthermore, one fewer value is computed,
  - thus potentially saving a register
  - and decreasing the possibility of spilling

Loop preparation

- Before attempting IVE, it is best to first perform :
  - constant propagation & constant folding
  - copy propagation
  - loop-invariant hoisting

How to do it, step 1

- First, find the basic IVs
  - scan loop body for defs of the form
    x = x + c or x = x – c
    where c is loop-invariant
  - record these basic IVs as
    x = (x, 1, c)
  - this represents the IV: x = x * 1 + c

Representing IVs

- Characterize all induction variables by:

  (base-variable, offset, multiple)
  - where the offset and multiple are loop-invariant
- IOW, after an induction variable is defined it equals:

  offset + multiple * base-variable

How to do it, step 2

- Scan for derived IVs of the form
  k = i * c1 + c2
  - where i is a basic IV,
    this is the only def of k in the loop, and
    c1 and c2 are loop invariant
- We say k is in the family of i
- Record as k = (i, c1, c2)

How to do it, step 3

- Iterate, looking for derived IVs of the form
  k = j * c1 + c2
  - where IV j =(i, a, b), and
  - this is the only def of k in the loop, and
  - there is no def of i between the def of j and the def of k
  - c1 and c2 are loop invariant
- Record as k = (i, a*c1, b*c1+c2)

Simple Example of IVE

```
    i <- 0
H:
    if i >= n goto exit
    j <- i * 4
    k <- j + a
    M[k] <- 0
    i <- i + 1
    goto H
```

```
i: (i, 1, 1) i.e., i = 1 + 1 * i
j: (i, 0, 4) i.e., j = 0 + 4 * i
k: (i, a, 4) i.e., k = a + 4 * i
```

So, j & k are in family of i

Finding the IVs

- Maintain three tables: basic & maybe & other
- Find basic Ivs:
  Scan stmts. If var $\notin$ maybe, and of proper form, put into basic. Otherwise, put var in other and remove from maybe.
- Find compound Ivs:
  - If var defined more than once, put into other
  - For all stmts of proper form that use a basic IV
    - » FIX THIS SLIDE

How to do it, step 4

- This is the strength reduction step
- For an induction variable k = (i, c1, c2)
  - initialize k = i * c1 + c2 in the preheader
  - replace k's def in the loop by
    k = k + c1
  - make sure to do this after i's def

How to do it, step 5

- This is the comparison rewriting step
- For an induction variable $k = (i, a_k, b_k)$
  - If k used only in definition and comparison
  - There exists another variable, j, in the same class and is not "useless" and $j=(i, a_j, b_j)$
- Rewrite $k < n$ as
  $j < (b_j/b_k)(n-a_k)+a_j$
- Note: since they are in same class:
  $(j-a_j)/b_j = (k-a_k)/b_k$

Notes

- Are the c1, c2 constant, or just invariant?
  - if constant, then you can keep folding them: they're always a constant even for derived IVs
  - otherwise, they can be expressions of loop-invariant variables
- But if constant, can find IVs of the type
  $x = i/b$
  and know that it's legal, if b evenly divides the stride…

Is it faster? (2)

- On some hardware, adds are much faster than multiplies
- But…not always a win!
  - Constant multiplies might otherwise be reduced to shifts/adds that result in even better code than IVE
  - Scaling of addresses (i*4) might come for free on your processor's address modes
- So maybe: only convert **`i*c1+c2`** when c1 is loop invariant but <u>not</u> a constant

Common loop optimizations

- Hoisting of loop-invariant computations
  - pre-compute before entering the loop
- Elimination of induction variables
  - change p=i*w+b to p=b,p+=w, when w,b invariant
- Loop unrolling
  - to to improve scheduling of the loop body
- Software pipelining
  - To improve scheduling of the loop body
- Loop permutation
  - to improve cache memory performance

Requires understanding data dependencies

Dependencies in Loops

- Loop independent data dependence occurs between accesses in the same loop iteration.
- Loop-carried data dependence occurs between accesses across different loop iterations.
- There is data dependence between
  access a at iteration i-k and
  access b at iteration i when:
  - a and b access the same memory location
  - There is a path from a to b
  - Either a or b is a write

Defining Dependencies

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

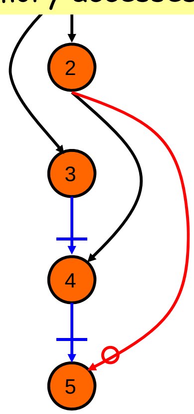
```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```


Example Dependencies

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

These are scalar dependencies. The same idea holds for memory accesses.

source	type	target	due to
S1	δ^f	S2	a
S1	δ^f	S3	a
S2	δ^f	S4	b
S3	δ^a	S4	d
S4	δ^a	S5	b
S2	δ^o	S5	b



What can we do with this information?
What are anti- and flow- called "false" dependences?

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.

δ^f loop carried

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

δ^f loop independent

Unroll Loop to Find Dependencies

δ^f loop carried

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

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

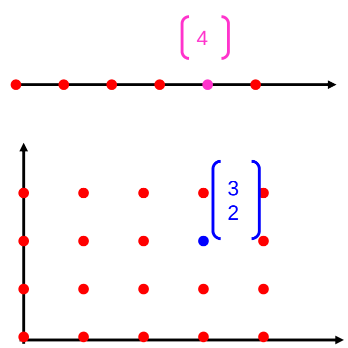
Distance/Direction of the dependence is also important.

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];
}
```

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

$d = I_t - I_s$

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

$I_s + d = I_t$

```
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

Example of Distance Vectors

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

j			
	$A_{0,2}=$ $=A_{0,2}$ $B_{0,3}=$ $=B_{0,2}$ $C_{1,2}=$ $=C_{0,3}$	$A_{1,2}=$ $=A_{1,2}$ $B_{1,3}=$ $=B_{1,2}$ $C_{2,2}=$ $=C_{1,3}$	$A_{2,2}=$ $=A_{2,2}$ $B_{2,3}=$ $=B_{2,2}$ $C_{3,2}=$ $=C_{2,3}$
	$A_{0,1}=$ $=A_{0,1}$ $B_{0,2}=$ $=B_{0,1}$ $C_{1,1}=$ $=C_{0,2}$	$A_{1,1}=$ $=A_{1,1}$ $B_{1,2}=$ $=B_{1,1}$ $C_{2,1}=$ $=C_{1,2}$	$A_{2,1}=$ $=A_{2,1}$ $B_{2,2}=$ $=B_{2,1}$ $C_{3,1}=$ $=C_{2,2}$
	$A_{0,0}=$ $=A_{0,0}$ $B_{0,1}=$ $=B_{0,0}$ $C_{1,0}=$ $=C_{0,1}$	$A_{1,0}=$ $=A_{1,0}$ $B_{1,1}=$ $=B_{1,0}$ $C_{2,0}=$ $=C_{1,1}$	$A_{2,0}=$ $=A_{2,0}$ $B_{2,1}=$ $=B_{2,0}$ $C_{3,0}=$ $=C_{2,1}$

i

Example of Distance Vectors

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

j			
	$A_{0,2}=$ $=A_{0,2}$ $B_{0,3}=$ $=B_{0,2}$ $C_{1,2}=$ $=C_{0,3}$	$A_{1,2}=$ $=A_{1,2}$ $B_{1,3}=$ $=B_{1,2}$ $C_{2,2}=$ $=C_{1,3}$	$A_{2,2}=$ $=A_{2,2}$ $B_{2,3}=$ $=B_{2,2}$ $C_{3,2}=$ $=C_{2,3}$
	$A_{0,1}=$ $=A_{0,1}$ $B_{0,2}=$ $=B_{0,1}$ $C_{1,1}=$ $=C_{0,2}$	$A_{1,1}=$ $=A_{1,1}$ $B_{1,2}=$ $=B_{1,1}$ $C_{2,1}=$ $=C_{1,2}$	$A_{2,1}=$ $=A_{2,1}$ $B_{2,2}=$ $=B_{2,1}$ $C_{3,1}=$ $=C_{2,2}$
	$A_{0,0}=$ $=A_{0,0}$ $B_{0,1}=$ $=B_{0,0}$ $C_{1,0}=$ $=C_{0,1}$	$A_{1,0}=$ $=A_{1,0}$ $B_{1,1}=$ $=B_{1,0}$ $C_{2,0}=$ $=C_{1,1}$	$A_{2,0}=$ $=A_{2,0}$ $B_{2,1}=$ $=B_{2,0}$ $C_{3,0}=$ $=C_{2,1}$

i

A yields: $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ B yields: $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ C yields: $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$

Data Dependences

Loop carried: between two statements instances in two different iterations of a loop.
Loop independent: between two statements instances in the same loop iteration.

Lexically forward: the source comes before the target .
Lexically backward: otherwise.

The right-hand side of an assignment is considered to precede the left-hand side.

Review of Linear Algebra
Lexicographic Order

Two n-vectors **a** and **b** are equal, **a** = **b**, if $a_i = b_i$, $1 \le i \le n$.

We say that **a** is less than **b**, **a**<**b**, if $a_i < b_i$, $1 \le i \le n$.

We say that **a** is lexicographically less than **b**, at level j, **a** $\ll_j$ **b**, if $a_i = b_i$, $1 \le i < j$ and $a_j < b_j$.

We say that **a** is lexicographically less than **b**, **a** $\ll$ **b**, if there is a j, $1 \le j \le n$, such that **a** $\ll_j$ **b**.

Lexicographic Order Example of vectors

Consider the vectors **a** and **b** below :

$$\mathbf{a} = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

We say that **a** is lexicographically less than **b** at level 3, $\mathbf{a} \prec_3 \mathbf{b}$, or simply that $\mathbf{a} \prec \mathbf{b}$.

Both **a** and **b** are lexicographically positive because $\mathbf{0} \prec \mathbf{a}$, and $\mathbf{0} \prec \mathbf{b}$.

Properties of Lexicographic Order

Let $n \geq 1$, and **i**, **j**, **k** denote arbitrary vectors in R^n

1. For each u in $1 \leq u \leq n$, the relation $\ll_u$ in R^n is irreflexive and transitive.
2. The n relations $\ll_u$ are pairwise disjoint: $\mathbf{i} \ll_u \mathbf{j}$ and $\mathbf{i} \ll_v \mathbf{j}$ imply that $u = v$.
3. If $\mathbf{i} \neq \mathbf{j}$, there is a unique integer u such that $1 \leq u \leq n$ and exactly one of the following two conditions holds:
   $\mathbf{i} \ll_u \mathbf{j}$ or $\mathbf{j} \ll_u \mathbf{i}$.
4. $\mathbf{i} \ll_u \mathbf{j}$ and $\mathbf{j} \ll_v \mathbf{k}$ together imply that $\mathbf{i} \ll_w \mathbf{k}$, where $w = \min(u,v)$.

Data Dependence in Loops

An Example

Find the dependence relations due to the array X in the program below:

```
(S1)   for i = 2 to 9 do
(S2)      X[i] = Y[i] + Z[i]
(S3)      A[i] = X[i-1] + 1
(S4)   end for
```

Solution

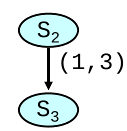
To find the data dependence relations in a simple loop, we can unroll the loop and see which statement instances depend on which others:

	i = 2	i = 3	i = 4
(s2)	X[2]=Y[2]+Z[2]	X[3] =Y[3]+Z[3]	X[4]=Y[4]+Z[4]
(s3)	A[2]=X[1]+1	A[3] =X[2]+1	A[4]=X[3]+1

From Wolfe

Data Dependence in Loops

```
(S1)  for i = 2 to
 9 do
 (S2)     X[i] =
Y[i] + Z[i]
 (S3)     A[i] =
X[i-1] + 1
 (S4) end for
```



Data dependence graph for statements in a loop
(1,3) := iteration distance is 1, latency is 3.

	i = 2	i = 3	i = 4
(s2)	X[2]=Y[2]+Z[2]	X[3]=Y[3]+Z[3]	X[4]=Y[4]+Z[4]
(s3)	A[2]=X[1]+1	A[3]=X[2]+1	A[4]=X[3]+1

There is a *loop-carried, lexically forward, flow dependence* from S_2 to S_3.

From Wolfe