Automatic Parallelization - Part 4

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

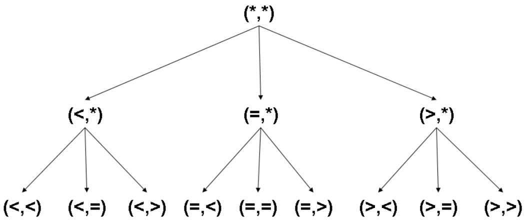
Data Dependence Framework

- Given two array references (with s dimensions and nested in loop nest of depth d):
  S_v : $X(f_1(I_1, ..., I_d),\ f_2(\bar{I}), ..., f_s(\bar{I}))$
  S_w : $X(g_1(I_1, ..., I_d),\ g_2(\bar{I}), ..., g_s(\bar{I}))$
  - We test for both $S_v\ \delta^* S_w$ and $S_w\ \delta^* S_v$ simultaneously
  - The particular type of dependence ($\delta,\ \bar{\delta},\ or\ \delta^o$) depends on the position of references (lhs or rhs) and the direction of dependence
- We first test to see if the array regions accessed by the two references intersect
  - Intersection will occur when the subscript functions are equal simultaneously

$$
\begin{aligned}
f_1(i_1, ..., i_d) &= g_1(j_1, ..., j_d) \\
f_2(i_1, ..., i_d) &= g_2(j_1, ..., j_d) \\
&... \\
f_s(i_1, ..., i_d) &= g_s(j_1, ..., j_d)
\end{aligned}
$$

Data Dependence Framework

- Test for intersection with a DV $(*, *, ..., *)$
- If *independence* can be proven with this DV, then the regions accessed by the two references are disjoint
- Otherwise, one "*" in the DV is refined to "<", "=", and ">", and testing is continued with these three refined DV
- Thus, testing is done by hierarchical expansion of one "*" at a time
- If independence can be proven at any point in the hierarchy, then the DV beneath it need not be tested



Complement and Product of Direction Vectors

- Complement of a DV $\Psi = (\Psi_1, ..., \Psi_d)$ is another DV $\Psi^{-1} = (\Psi_1^{-1}, ..., \Psi_d^{-1})$, where each Ψ_k^{-1} is computed from Ψ_k as follows

Ψ_k	$< \; = \; > \; \leq \; \geq \; \neq \; *$
Ψ_k^{-1}	$> \; = \; < \; \geq \; \leq \; \neq \; *$

- Product of two DVs $\Psi^1 = (\Psi_1^1, ..., \Psi_d^1)$ and $\Psi^2 = (\Psi_1^2, ..., \Psi_d^2)$ is defined to be $\Psi = (\Psi_1, ..., \Psi_d) = \Psi^1 \times \Psi^2$, where $\Psi_1 = \Psi_1^1 \times \Psi_1^2$, $\Psi_2 = \Psi_2^1 \times \Psi_2^2$,..., $\Psi_d = \Psi_d^1 \times \Psi_d^2$, and $\times$ is defined on DV elements as follows
- "$\cdot$" means a null DV element

Product of DV elements

X	<	=	>	≤	≥	≠	*
<	<	.	.	<	.	<	<
=	.	=	.	=	=	.	=
>	.	.	>	.	>	>	>
≤	<	=	.	≤	=	<	≤
≥	.	=	>	=	≥	>	≥
≠	<	.	>	<	>	≠	≠
*	<	=	>	≤	≥	≠	*

Data Dependence Framework

- Compute product of different DV corresponding to various subscripts to get one DV
  $\Psi = \Psi_1 \times \Psi_2 \times \ldots \times \Psi_s$
- If this combination produces any "·" entries, then there is no simultaneous intersection at all and so there can be no dependence
- To get the data dependence DV, we must intersect Ψ with the execution order DV: $\Psi_{v \rightarrow w} = \Psi \times \Omega_{v \rightarrow w}$
- If this produces any "·" entries, there is no dependence from S_v to S_w

Data Dependence Framework

- If all the entries are valid, we add the data dependence relation: $S_v \ \delta^*_{v \to w} \ S_w$ to the DDG
- The actual type of dependence ($\delta, \ \overline{\delta}, \ or \ \delta^o$) will depend on the position of the references
- To check dependence from S_w to S_v, we compute: $\Psi_{w \to v} \ = \ \Psi^{-1} \ \times \ \Omega_{w \to v}$
- If all the entries are valid, we add the data dependence relation: $S_w \ \delta^*_{w \to v} \ S_v$ to the DDG

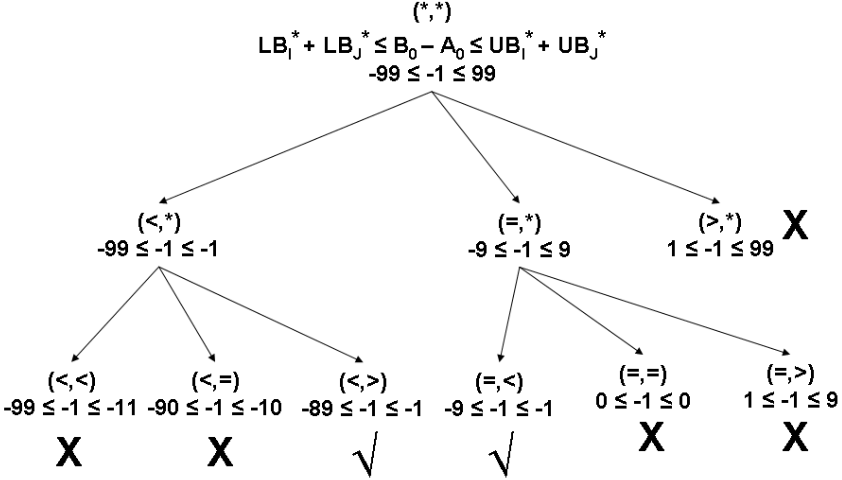
Data Dependence Framework Test Example - 2.1

Given program:

```
       for I = 1 to 10 do {
         for J = 1 to 10 do {
S1:        A(I*10+J) = ...
S2:          ... = A(I*10+J-1)
         }
       }
```

- Dependence equation: $10I_1 + J_1 - 10I_2 - J_2 \ = \ -1$
- GCD Test with (*,*): GCD(10, 1, -10, -1) divides -1, which is true and hence dependence exists. Now we need to apply Banerjee's test
- $$\begin{array}{llll} LB_I^* = -90 & LB_I^< = -90 & LB_I^= = 0 & LB_I^> = 10 \\ UB_I^* = 90 & UB_I^< = -10 & UB_I^= = 0 & UB_I^> = 90 \\ LB_J^* = -9 & LB_J^< = -9 & LB_J^= = 0 & LB_J^> = 1 \\ UB_J^* = 9 & UB_J^< = -1 & UB_J^= = 0 & UB_J^> = 9 \end{array}$$

Data Dependence Framework Test Example - 2.2



Data Dependence Framework Test Example - 2.3

- The dependence test returns two DVs: (<,>) and (=,<)
- There is only one subscript
- Recall that $S1\ \Theta_{(=,\leq)} S2$, $S2\ \Theta_{(=,<)} S1$, $S1\ \Theta_{(<,*)} S2$, $S2\ \Theta_{(<,*)} S1$, are all possible
- Intersect these with the execution order DVs
  $(<,>)\ \times\ (<,*)\ =\ (<,>)$
  $(=,<)\ \times\ (=,\leq)\ =\ (=,<)$
  Other products produce "·" values
- Therefore we get: $S1\ \delta_{(=,<)} S2$ and $S1\ \delta_{(<,>)} S2$
- There is no need to test $S2\ \delta^*\ S1$, since not all entries are "·"

Concurrentization or Parallelization

- If all the dependence relations in a loop nest have a direction vector value of "=" for a loop, then the iterations of that loop can be executed in parallel with no synchronization between iterations
- Any dependence with a forward (<) direction in an outer loop will be satisfied by the serial execution of the outer loop
- If an outer loop L is run in sequential mode, then all the *dependences* with a forward (<) direction at the outer level (of L) will be automatically satisfied (even those of the loops inner to L)
- However, this is not true for those dependences with (=) direction at the outer level; the dependences of the inner loops will have to be satisfied by appropriate statement ordering and loop execution order

Concurrentization Examples

```
        for I = 2 to N do {
          for J = 2 to N do {
S1:         A(I,J) = B(I,J) + 2
S2:         B(I,J) = A(I-1, J-1) – B(I,J)
          }
        }
```

S1 $\delta_{(<,<)}$ S2, S1 $\overline{\delta}_{(=,=)}$ S2, S2 $\overline{\delta}_{(=,=)}$ S2

	I = 1	I = 2
J = 1	A(2,2)= = A(1,1)	A(3,2)= = A(2,1)
J = 2	A(2,3)= = A(1,2)	A(3,3)= = A(2,2)
J = 3	A(2,4)= = A(1,3)	A(3,4)= = A(2,3)

If the I loop is run in serial mode then, the J loop can be run in parallel mode

```
        for I = 2 to N do {
          for J = 2 to N do {
S1:         A(I,J) = B(I,J) + 2
S2:         B(I,J) = A(I, J-1) – B(I,J)
          }
        }
```

S1 $\delta_{(=,<)}$ S2, S1 $\overline{\delta}_{(=,=)}$ S2, S2 $\overline{\delta}_{(=,=)}$ S2

	I = 1	I = 2
J = 1	A(2,2)= = A(2,1)	A(3,2)= = A(3,1)
J = 2	A(2,3)= = A(2,2)	A(3,3)= = A(3,2)
J = 3	A(2,4)= = A(2,3)	A(3,4)= = A(3,3)

The J loop cannot be run in parallel mode. However, the I loop can be run in parallel mode

Loop Transformations for increasing Parallelism

- Recurrence breaking
  - Ignorable cycles
  - Scalar expansion
  - Scalar renaming
  - Node splitting
  - Threshold detection and index set splitting
  - If-conversion
- Loop interchanging
- Loop fission
- Loop fusion

Ignorable Cycles

- Any single statement recurrence based on $\overline{\delta}$ may be ignored
- The program:

```
  for I = 2 to 100 do {
S:   X(I-1) = F(X(I))
  }
```

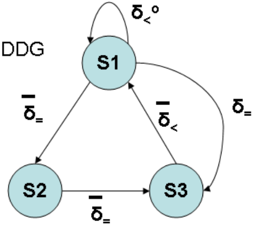
has the dependence $S \, \overline{\delta} \, S$, but it can be vectorized as follows:
X(1:99) = F(X(2:100))

Scalar Expansion

Not vectorizable or parallelizable

```
for I = 1 to N do {
  S1:   T = A(I)
  S2:   A(I) = B(I)
  S3:   B(I) = T
}
```

Cyclic DDG



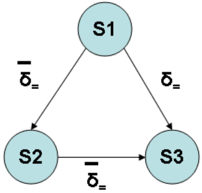
Vectorizable due to scalar expansion

```
for I = 1 to N do {
  S1:   Tx(I) = A(I)
  S2:   A(I) = B(I)
  S3:   B(I) = Tx(I)
}
```

Parallelizable due to privatization

```
forall I = 1 to N do {
  private temp
  S1:   temp = A(I)
  S2:   A(I) = B(I)
  S3:   B(I) = temp
}
```

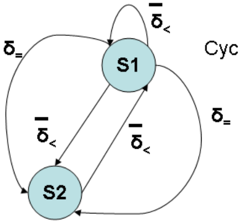
Acyclic DDG



Scalar Expansion is not always profitable

Not vectorizable or parallelizable

```
for I = 1 to N do {
  S1:    T = T + A(I) + A(I+2)
  S2:    A(I) = T
}
```

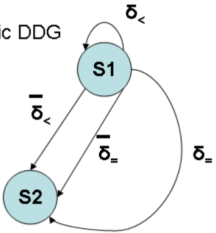


Cyclic DDG

Not vectorizable even
after scalar expansion

```
for I = 1 to N do {
  S1:    Tx(I) = Tx(I-1)+A(I)+A(I+2)
  S2:    A(I) = Tx(I)
}
```

Still cyclic DDG



Scalar Renaming

The output dependence
S1 δ^o S3 cannot be broken
by scalar expansion

1.

```
for I = 1 to N do {
  S1:   T = A(I) + B(I)
  S2:   C(I) = T*2
  S3:   T = D(I) * B(I)
  S4:   A(I+2) = T + 5
}
```

The output dependence
S1 δ^o S3 CAN be broken
by scalar renaming

2.

```
for I = 1 to N do {
  S1:   T1 = A(I) + B(I)
  S2:   C(I) = T1*2
  S3:   T2 = D(I) * B(I)
  S4:   A(I+2) = T2 + 5
}
```

3.

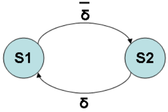
```
S3:   T2(1:100) = D(1:100) * B(1:100)
S4:   A(3:102) = T2(1:100) + 5(1:100)
S1:   T1(1:100) = A(1:100) + B(1:100)
S2:   C(1:100) = T1(1:100)*2(1:100)
       T = T2(100)
```

5(1:100) and 2(1:100)
are vectors of constants

Node Splitting

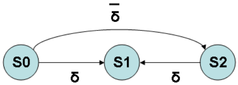
- Node splitting can be used in breaking a cycle consisting of an anti-dependence, but this introduces new temporary arrays

```
for I = 1 to 100 do {
S1:   A(I) = X(I+1) + X(I)
S2:   X(I+1) = B(I)
}
```



δ cannot be removed

```
for I = 1 to 100 do {
S0:   T(I) = X(I+1)
S1:   A(I) = T(I) + X(I)
S2:   X(I+1) = B(I)
}
```



This loop can be vectorized

```
S0:   T(1:100) = X(2:101)
S2:   X(2:101) = B(1:100)
S1:   A(1:100) = T(1:100) + X(1:100)
```

Thresholds

```
for I = 1 to 100 do {
    X(I+5) = X(I)
}
```

Cannot be vectorized
Threshold value = 5

```
for I = 1 to 20 do {
  for J = 1 to 5 do {
    X(I*5+J) = X(I*5+J-5)
  }
}
```

```
for I = 1 to 20 do {
  X(5*I+1 : 5*I+5)
    = X(5*I-4 : 5*I)
```

Thresholds can be found by modifications of Banjerjee's test

Cannot be vectorized
Threshold value = 50

```
for I = 1 to 100 do {
   A(I) = A(101- I)
}
```

```
for J = 1 to 2 do {
  A(50*J – 49 : 50*J)
    = A(150 - 50*I : 101-50*I)
```

If-Conversion

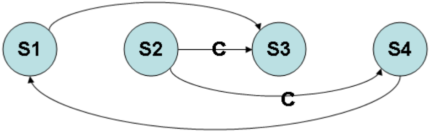
```
for I = 1 to 100 do {
  if (A(I) <= 0) then contnue
  A(I) = B(I) + 3
}
```

```
for I = 1 to 100 do {
  BR(I) = (A(I) <= 0)
  if (~ BR(I))  then
     A(I) = B(I) + 3
}
```

```
BR(1:N) = (A(1:N) <= 0)
where (~ BR(1:N))
   A(1:N) = B(1:N) + 3
```

```
         for I = 1 to N do {
S1:        A(I) = D(I) + 1
S2:        if (B(I) > 0) then
S3:          C(I) = C(I) + A(I)
S4:          D(I+1) = D(I+1) + 1
           end if
         }
```

```
         for I = 1 to N do {
S2:        temp(1:N) = B(1:N) > 0
S4:        where (temp(1:N))
               D(2:N+1) = D(2:N+1) + 1
S1:        A(1:N) = D(1:N) + 1
S3:        where (temp(1:N))
               C(1:N) = C(1:N) + A(1:N)
         }
```



Loop Interchange

- For machines with vector instructions, loops can be interchanged to find vector operations, if the original inner loop cannot be vectorized
- For multi-core and multi-processor machines, parallel outer loops are preferred and loop interchange may help to make this happen
- Requirements for simple loop interchange
  1. The loops L1 and L2 must be tightly nested (no statements between loops)
  2. The loop limits of L2 must be invariant in L1
  3. There are no statements S_v and S_w (not necessarily distinct) in L1 with a dependence $S_v \ \delta^*_{(<,>)} \ S_w$

Loop Interchange for Vectorizability

```
for I = 1 to N do {
   for J = 1 to N do {
S:       A(I,J+1) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Inner loop is not vectorizable

$S\ \delta_{(=,<)}\ S$

```
for J = 1 to N do {
   for I = 1 to N do {
S:       A(I,J+1) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Inner loop is vectorizable

$S\ \delta_{(<,=)}\ S$

```
for J = 1 to N do {
S:       A(1:N, J+1) = A(1:N, J) * B(1:N, J) + C(1:N, J)
}
```

Loop Interchange for parallelizability

```
for I = 1 to N do {
   for J = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Outer loop is not parallelizable, but inner loop is

$S\ \delta_{(<,=)}\ S$

Less work per thread

```
for J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

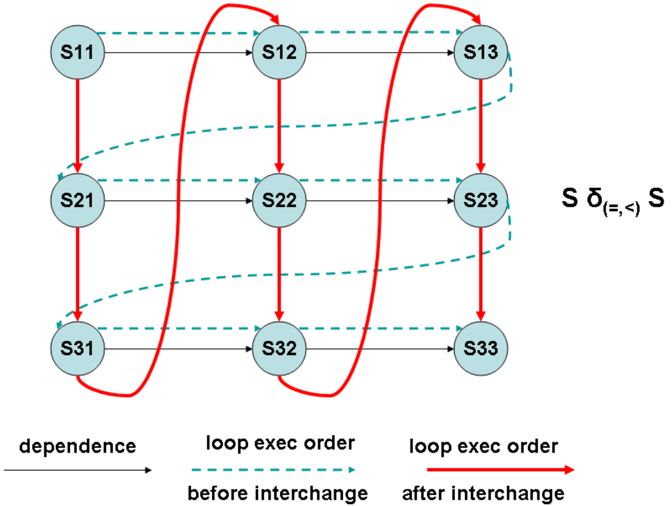
Outer loop is parallelizable but inner loop is not

$S\ \delta_{(=,<)}\ S$

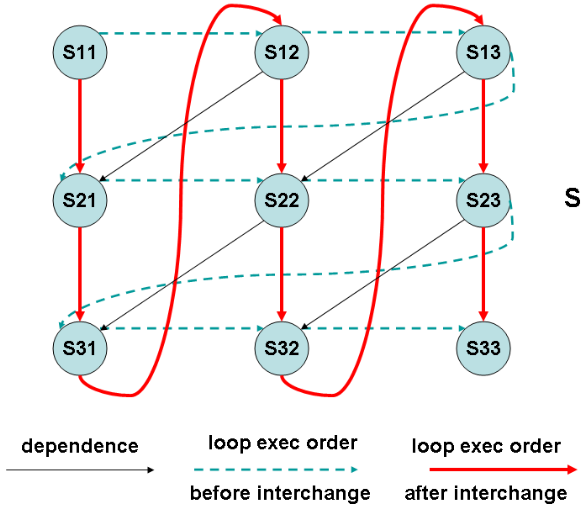
More work per thread

```
forall J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Legal Loop Interchange

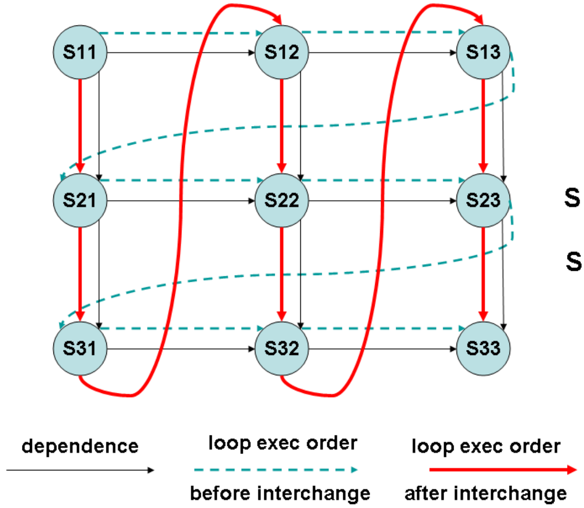


Illegal Loop Interchange



$S\ \delta_{(<,>)}\ S$

Legal but not beneficial Loop Interchange



$S\ \delta_{(=,<)}\ S$
and
$S\ \delta_{(<,=)}\ S$

Loop Fission - Motivation

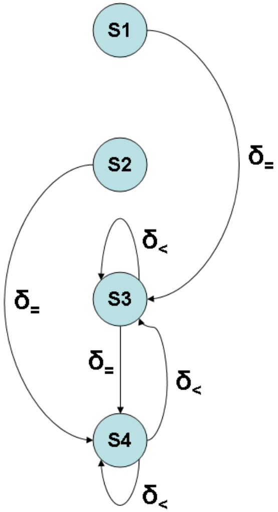
```
      for I = 1 to N do {
S1:     A(I) = E(I) + 1
S2:     B(I) = F(I) * 2
S3:     C(I+1) = C(I) * A(I) + D(I)
S4:     D(I+1) = C(I+1) * B(I) + D(I)
      }
```

The above loop cannot be vectorized

```
L1:   for I = 1 to N do {
S1:     A(I) = E(I) + 1
S2:     B(I) = F(I) * 2
      }

L2:   for I = 1 to N do {
S3:     C(I+1) = C(I) * A(I) + D(I)
S4:     D(I+1) = C(I+1) * B(I) + D(I)
      }
```

L1 can be vectorized, but L2 cannot be

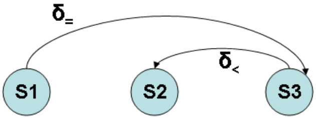


Loop Fission Lemma

- *Lemma*: If a loop L contains statements S_k and S_j, where S_K follows S_j in the loop and $S_k \ \delta^*_< \ S_j$, then loop fission may not split the loop at any point between S_j and S_k
- Loop fission may not be used to break a cycle of dependence into separate loops

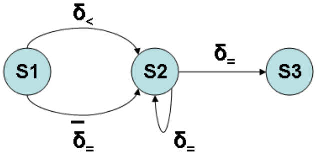
Loop Fission: Legal and Illegal

```
        for I = 1 to N do {
S1:         A(I) = D(I) * T
S2:         B(I) = (C(I) + E(I))/2
S3:         C(I+1) = A(I) + 1
        }
```



In the above loop, S3 $\delta_<$ S2, and S3 follows S2. Therefore, cutting the loop between S2 and S3 is illegal. However, cutting the loop between S1 and S2 is legal.

```
        for I = 1 to N do {
S1:         A(I+1) = B(I) +D(I)
S2:         B(I) = (A(I) + B(I))/2
S3:         C(I) = B(I) + 1
        }
```



The above loop can be cut between S1 and S2, and also between S2 and S3

Conditions for Loop Fusion

- Same index sets
- Loops must be adjacent
- No conditional branch (that exits) in either loop (unless the conditions are identical)
- I/O in *both* loops makes fusion illegal, but I/O in one of the loops is permitted
- Data dependence requirement (later)

Same Index sets for loop fusion

LOOP 1

```
     for I = 1 to N do {
S1:     A(I) = B(I) + C(I)
     }
```

LOOP 2

```
     for I = 2 to N do {
S2:     D(I) = E(I) * 2
     }
```

The above loops are not fusible

Option for LOOP 1

```
     A(1) = B(1) + C(1)
     for I = 2 to N do {
S1:     A(I) = B(I) + C(I)
     }
```

Options for LOOP 2

Option A

```
     for I = 1 to N do {
S2:     D(I+1) = E(I+1) * 2
     }
```

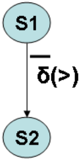
Option B

```
     for I = 1 to N do {
S2:    if (I >= 2)
           D(I) = E(I) * 2
     }
```

Illegal loop fusion

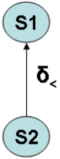
```
      for I = 1 to N do {
S1:       A(I) = B(I) + C(I)
      }

      for I = 1 to N do {
S2:       B(I+1) = D(I) * 2
      }
```



If the two loops are fused,
then the dependences change!

```
      for I = 1 to N do {
S1:       A(I) = B(I) + C(I)
S2:       B(I+1) = D(I) * 2
      }
```



Augmented Direction Vector

- Let S1 be a statement enclosed in a loop L1 with index set i, and let S2 be a statement enclosed in a loop L2 with index set j, and let the two index sets be identical. Let X be one of $\{\delta,\ \overline{\delta},\ \delta^o\}$ and let $S1\ X\ S2$.
- We define the *augmented DV* to be (?) where, $? \ \in\ \{<, =, >\}$ and we say $S1\ X(?)\ S2$ when
  1. there exist particular iterations of S1 and S2, say, $S1(i')$ and $S2(j')$ with $S1(i')\ X\ S2(j')$ and
  2. $i'\ ?\ j'$
- Definition 1 above allows a DV to have positions for loops that do not contain both S1 and S2
- *Lemma*: Let L1 and L2 be loops as above. If there are any statements S_j in L1 and S_k in L2 with $S_j\ \delta^*(>)\ S_k$, then fusing the loops is illegal

Augmented DV example

```
        for I = 2 to N do {
S1:         A(I) = D(I) * 2
        }

        for I = 2 to N do {
S2:         B(I) = A(I) + 1
        }
```

S1 δ(=) S2

```
        for I = 2 to N do {
S1:         A(I) = D(I) * 2
        }

        for I = 2 to N do {
S2:         B(I) = A(I-1) + 1
        }
```

S1 δ(<) S2

```
        for I = 2 to N do {
S1:         A(I) = D(I) * 2
        }

        for I = 2 to N do {
S2:         B(I) = A(I+1) + 1
        }
```

S1 $\bar{\delta}$(>) S2