CS 526

Advanced
Compiler
Construction

https://charithm.web.illinois.edu/cs526/sp2024/
(slides adapted from Sasa and Vikram)

Roadmap

- Control-flow Analysis
- SSA-based Analysis
- General Data-flow Analysis
- Dependence Analysis
- Pointer Analysis
- Interprocedural Analysis

And then some advanced topics: Vectorization, Tensor Compilation, ML in compilers

DEPENDENCE ANALYSIS

The slides based on lectures by Vikram Adve and David Padua and Dragon Book

Theme

How can a compiler enhance **parallelism** and **locality** in programs with arrays?

Exposing available parallelism is ***not easy***!

- Find parallel tasks
- Minimize communication & synchronization overhead

Data locality: A program has good data locality if CPU accesses the same data it has ***used recently*** (temporal locality) or ***data neighboring*** such data (spatial locality)

Theme

Parallelism and data locality go hand-in-hand

- Identify data locality $\Rightarrow$ know the parallelism

Previous data-flow analysis does not work

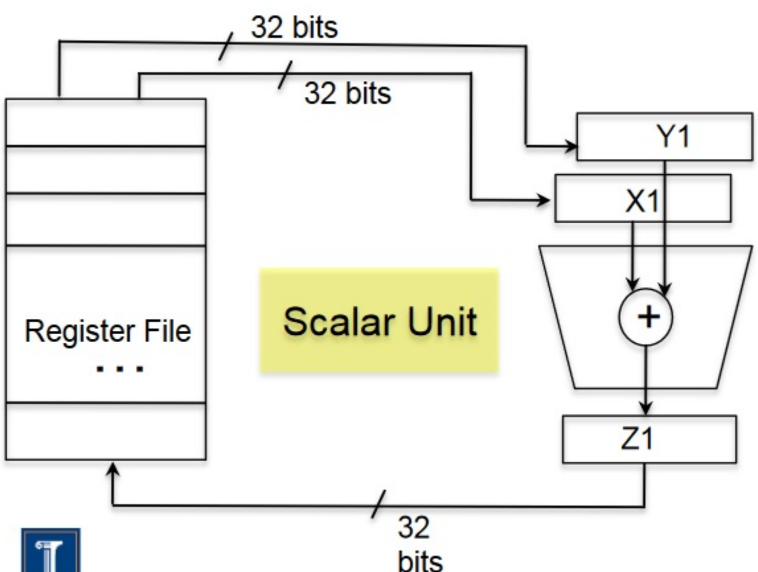
- We don't distinguish the ways the statement was reached, i.e. different executions of the same statement in the loop
- We didn't discuss how to treat arrays in that framework
- For parallelization we need to reason about the different dynamic executions of the same statement

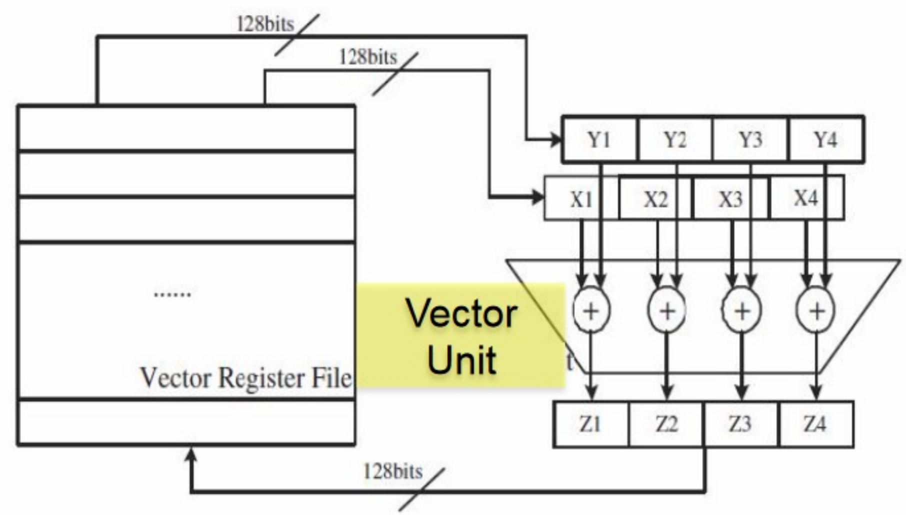
Motivation: Vectorization

n times
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st r3, addr3

```
for (i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

n/4 times
ldv vr1, addr1
ldv vr2, addr2
addv vr3, vr1, vr2
stv vr3, addr3



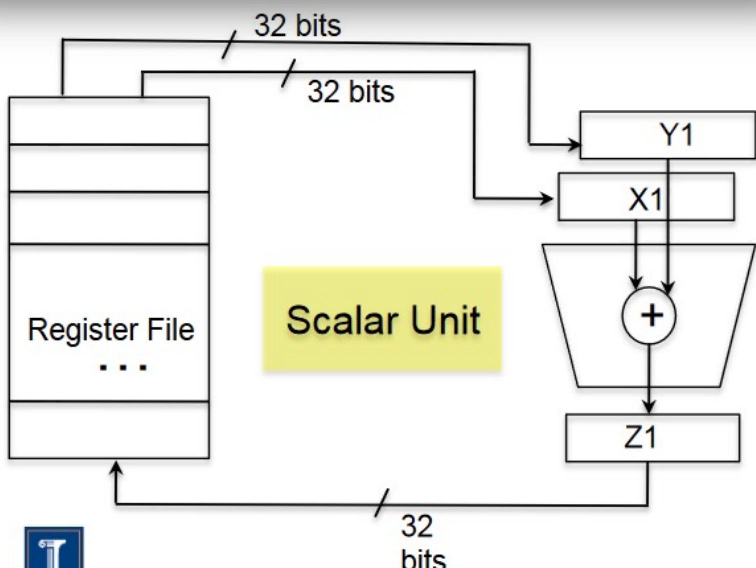


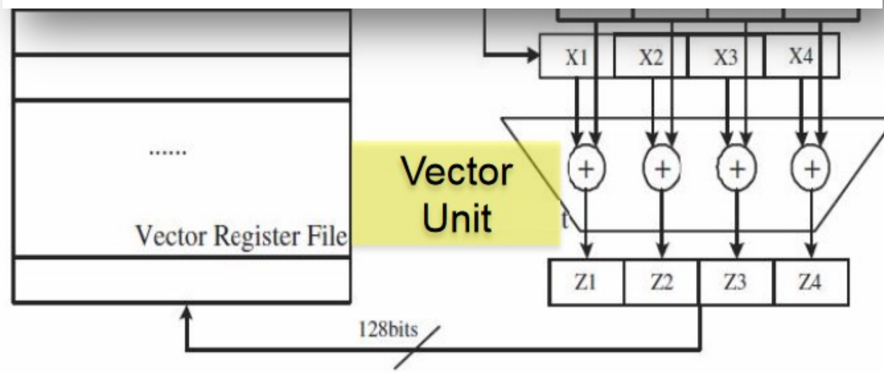
*Slide from Maria Garzaran and David Padua

Motivation: Vectorization

```
void vec_eltwise_product(vec_t* a, vec_t* b,
                         vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

```
void vec_eltwise_product_avx(vec_t* a, vec_t* b,
                             vec_t* c) {
    size_t i;
    __m256 va;
    __m256 vb;
    __m256 vc;
    for (i = 0; i < a->size; i += 8) {
        va = _mm256_loadu_ps(&a->data[i]);
        vb = _mm256_loadu_ps(&b->data[i]);
        vc = _mm256_mul_ps(va, vb);
        _mm256_storeu_ps(&c->data[i], vc);
    }
}
```





*Slide from Maria Garzaran and David Padua
** AVX code from Intel's Software&Services Group talk

Motivation: Task Parallelization

```
for (i=0; i < N; i++)
{
   Y[i] = X[i] - 1
   Y[i] = Y[i] * Y[i]
}
```

```
for (i=0; i < N/4; i++)
{
   Y[i] = X[i] - 1
   Y[i] = Y[i] * Y[i]
}
```

```
for (i=N/4; i < N/2; i++)
{
   Y[i] = X[i] - 1
   Y[i] = Y[i] * Y[i]
}
```

```
for (i=N/2; i < 3*N/4; i++)
{
   Y[i] = X[i] - 1
   Y[i] = Y[i] * Y[i]
}
```

```
for (i=3*N/4; i < N; i++)
{
   Y[i] = X[i] - 1
   Y[i] = Y[i] * Y[i]
}
```

```
Wait for all threads to finish before proceeding
```

SPMD = Single program multiple data; there is a synchronization barrier at the end

Data Dependence

A **data dependence** from statement **S1** to statement **S2** exists if

1. there is a ***feasible execution path*** from S1 to S2, and
2. an instance of S1 ***references the same memory location*** as an instance of S2 in some execution of the program, and
3. at ***least one of the references is a store***.

Kinds of Data Dependence

Direct Dependence

X =…
…= X + …

Anti-dependence

… = X
X = …

Output Dependence

X = …
X = …

Dependence Graph

A **dependence graph** is a graph with:

- Each ***node represents a statement***, and
- Each ***directed edge*** from S1 to S2, if there is a **data dependence** between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).
    - S1 is known as a ***source*** node
    - S2 is known as a ***sink*** node

Kinds of Data Dependence

Dependence Graph Edges

Direct Dependence

S1: X =…
S2: … = X + …

$$S_1 \longrightarrow S_2$$

Anti-dependence

S1: … = X
S2: X = …

$$S_1 \,{+}\!\!\!\longrightarrow S_2$$

Output Dependence

S1: X = …
S2: X = …

$$S_1 \,{\circ}\!\!\!\longrightarrow S_2$$

Dependence Graph for Loops

(***Repeat***) A **dependence graph** is a graph with:

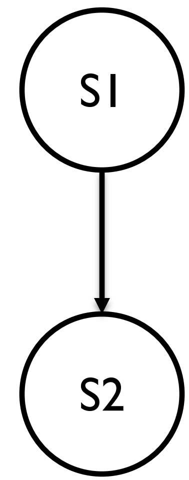
- one node per statement, and
- a directed edge from S1 to S2 if there is a data dependence between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).

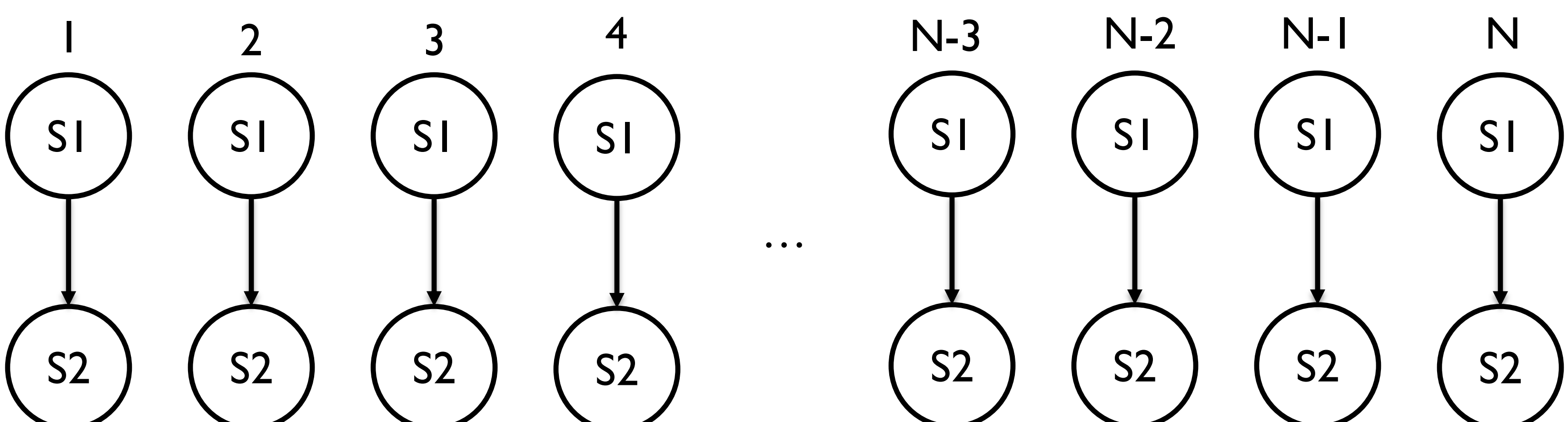
For loops: dependence graph is a ***summary of unrolled dependencies*** for different iterations

- Some (detailed) information may be lost

Dependence in Loops

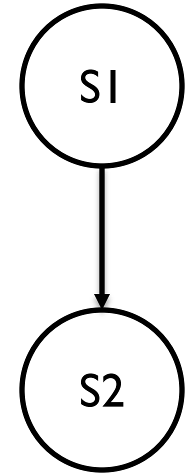
```
        int X[], Y[], a[], i;
        for i = 1 to N
S1:         X[i] = a[i] + 2
S2:         Y[i] = X[i] + 1
        end
```

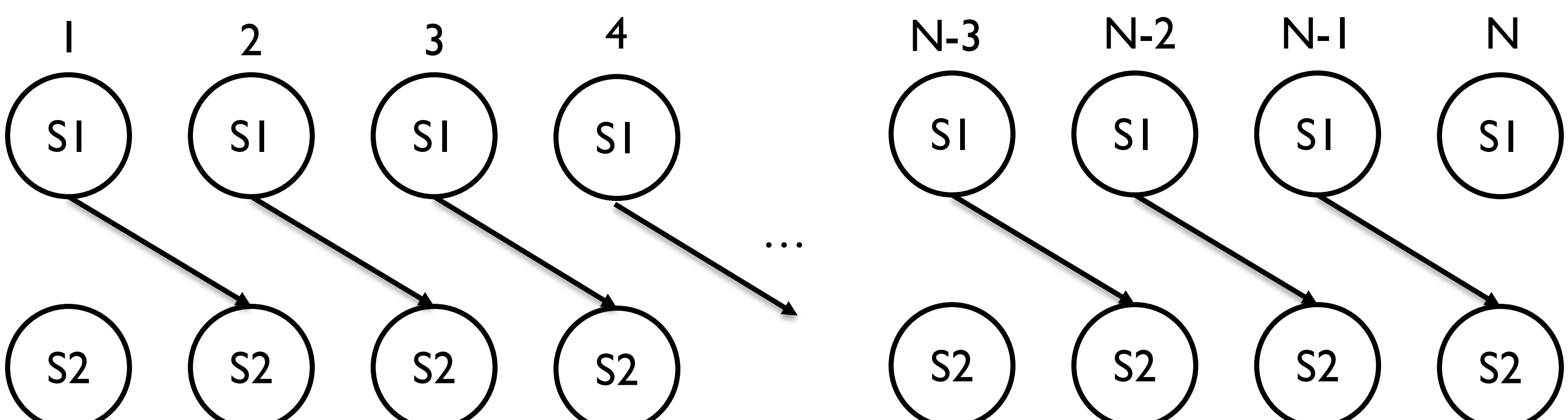




Dependence in Loops

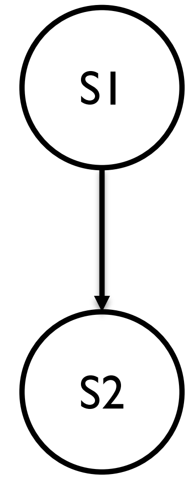
```
      int X[], Y[], a[], i;
      for i = 1 to N
S1:       X[i+1] = a[i] + 2
S2:       Y[i] = X[i] + 1
      end
```

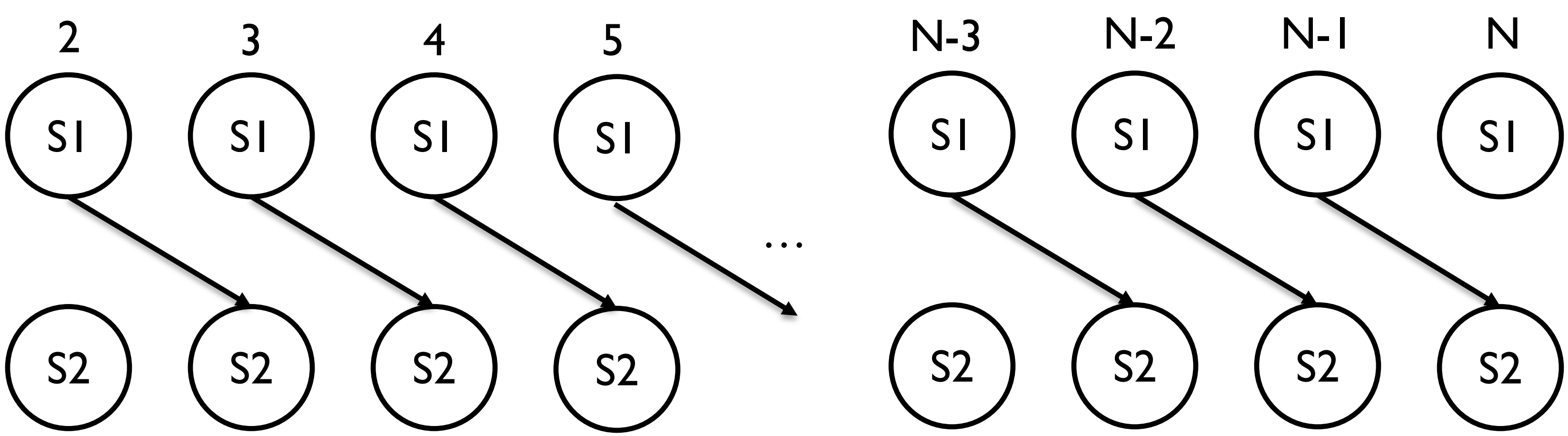




Dependence in Loops

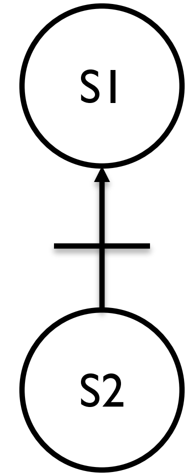
```
        int X[], Y[], a[], i;
        for i = 2 to N
S1:         X[i] = a[i] + 2
S2:         Y[i] = X[i-1] + 1
        end
```

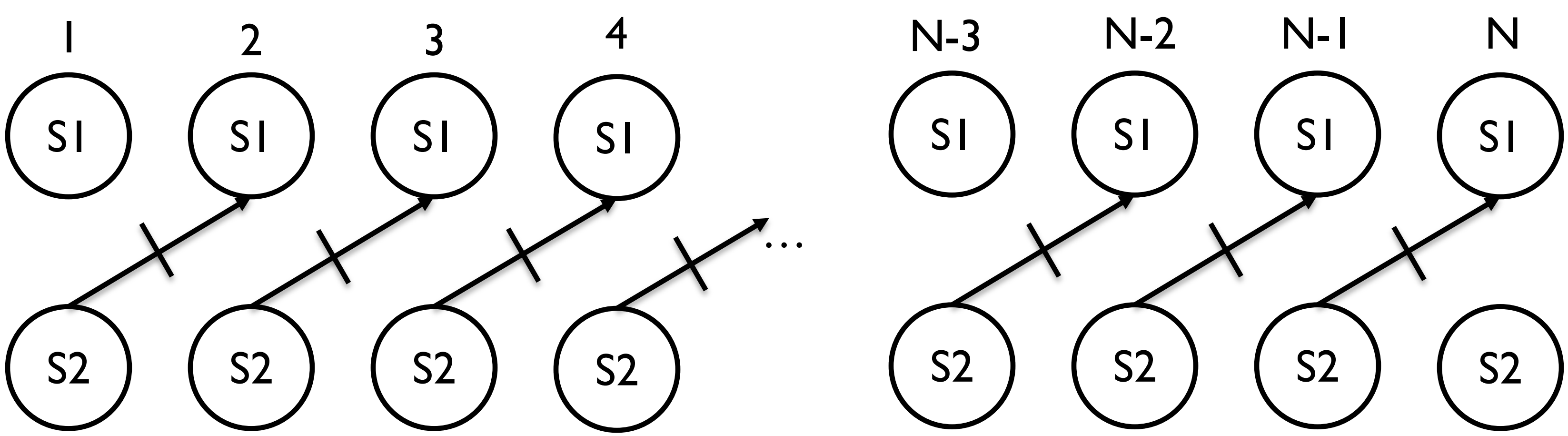




Dependence in Loops

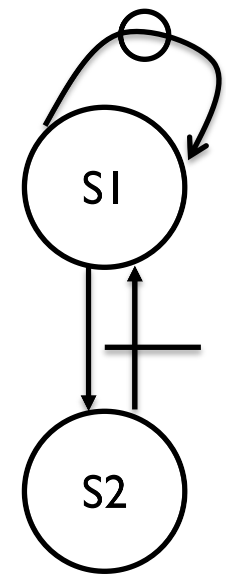
```
        int X[], Y[], a[], i;
        for i = 1 to N
S1:         X[i] = a[i] + 2
S2:         Y[i] = X[i+1] + 1
        end
```





Dependence in Loops

```
        int X[], Y[], a[], t, i;
        for i = 1 to N
S1:         t = a[i] + 2
S2:         Y[i] = t + 1
        end
```



Loop Carried Dependence: one that crosses the loop iteration boundary

Loop Independent Dependence: one that remains within the statements in the single iteration

Next…

Let us introduce the **affine transform theory**

- Reorder statements instead of remove
- Lets us use standard mathematical tools (solving linear equations, mathematical programming, solving linear constraints)

Reordering Transformation

Reordering Transformation: **merely changes the order** of execution of computations in a program, **without adding or deleting** executions of any computations.

Preserving Dependence: a reordering transformation preserves a dependence if it **preserves the relative execution order** of the ***source*** and ***sink*** statements of the dependence.

Reordering Transformation

Definition. Legal Transformation preserves the meaning of that program, i.e., **all externally visible outputs are identical to the original program**, and in identical order.

- We consider two programs equivalent (i.e., the transformation preserving the program meaning) if on the same inputs both the original and transformed programs, after being executed, produce the same outputs.

Theorem. A ***reordering*** transformation that preserves all data dependences in a program is a ***legal*** transformation.

Proof of Theorem 1

(by contradiction)

Loop-free program:

Let $S_1, \ldots S_n$ be the original execution order, and $i_1 \ldots i_n$ a permutation of the statement indices in the reordered program. If we reorder code without violating dependencies, but the output changed, then at least one statement would need to produce a different output. Since the statement is the same as in the original program, then its error must have propagated from the inputs. But in that case, there must have been a previous statement that violated (flow, anti, or output) dependence. Contradiction!

Loops:

The previous argument directly extends, by unrolling (and the index of the loop iteration represents the part of the permutation index).

Conditionals:

If there are conditional statements, the theorem must include ***control dependences*** in addition to data dependences.

(We will come back to this point next week)

Dependence in Loop Nests

Goal: Supporting transformations of a given loop nest (Assume perfect loop nest here)

Canonical Loop Nest: A loop nest is in canonical form if both lower bound and step of each loop are +1.

```
do i1 = 1 to n1
   do i2 = 1 to n2
      . . .
      do ik = 1 to nk
         statements
      enddo
      . . .
   enddo
enddo
```

Rectangular Loop Nest: The value of `n1` to `nk` does not change during the execution

Dependence in Loop Nests

```
do i1 = 1 to n1
   do i2 = 1 to n2
     . . .
      do ik = 1 to nk
        statements
      enddo
     . . .
   enddo
enddo
```

Iteration space

The *iteration space* of the loop nest is a set of points in a k-dimensional integer space (i.e., a polyhedron):

$$L = \{[i_1, \ldots, i_n] : 1 \le i_1 \le n_1 \wedge \ldots \wedge 1 \le i_k \le n_k\}$$

Each element $[i_1, \ldots, i_n]$ is an **iteration vector**

Example

```
for i in 0 to 5
    for j in i to 7
      Z[i,j] = i+j;
```

Inequalities:

$\mathbf{0 \le \boldsymbol{i}}$	$\boldsymbol{i} + \mathbf{0} \cdot \boldsymbol{j} + \mathbf{0} \ge \mathbf{0}$
$\boldsymbol{i} \le \mathbf{5}$	$-\boldsymbol{i} + \mathbf{0} \cdot \boldsymbol{j} + \mathbf{5} \ge \mathbf{0}$
$\boldsymbol{i} \le \boldsymbol{j}$	$-\boldsymbol{i} + \boldsymbol{j} + \mathbf{0} \ge \mathbf{0}$
$\boldsymbol{j} \le \mathbf{7}$	$\mathbf{0} \cdot \boldsymbol{i} - \boldsymbol{j} + \mathbf{7} \ge \mathbf{0}$

Turn the inequalities in the form $\boldsymbol{\alpha} \cdot \boldsymbol{i} + \boldsymbol{\beta} \cdot \boldsymbol{j} + \boldsymbol{\gamma} \ge \mathbf{0}$

- $[\alpha, \beta]$ become rows in the matrix **B**
- γ becomes an element in the vector **b**

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \ge \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Iteration of the Loop Nest

$$\{i \in \mathbb{Z}^n \mid Bi + b \geq 0\}$$

- n is the depth of the loop nest
- B is a $\mathrm{m}\times n$ matrix
- b is a vector with length m
- 0 is a vector of m zeros

Represent a convex polyhedron

Incorporating Symbolic Constraints (e.g., for `i < n`): add symbolic variable, extending the vector:

$$\left\{i \in \mathbb{Z} \quad \middle| \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ n \end{bmatrix} i + b \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}$$

Dependence in Loop Nests

Lexicographic Order: for iteration vectors
$I = [i_1, \ldots, i_n]$ and $I' = [i'_1, \ldots, i'_n]$:

$[i_1, \ldots, i_n] < [i'_1, \ldots, i'_n]$ iff there is a subscript **k**, such that $i_1 = i'_1$, ... $i_{k-1} = i'_{k-1}$ but $i_k < i'_k$

If $[i_1, \ldots, i_n] < [i'_1, \ldots, i'_n]$ we say that the iteration I **precedes** the iteration I'

Examples: [1,2,3] < [1,2,4] and [1, 2, 3] < [1,3,1]

Dependence in Loop Nests

```
do v1 = 1 to n1
    do v2 = 1 to n2
      . . .
        do vk = 1 to nk
            X[f1(I), …, fk(I)] = ...
            ... = X[g1(I’), …, gk(I’)]
        end
      . . .
    end
end
```

```
I=[v1,v2, …,vk]
I’=[v1’,v2’,…, vk’]
```

A data dependence exists iff $\exists$ `I,I’` $\in$ `[1..n1]`x…x `[1..nk]`
s.t. `[f1(I), …, fk(I)] = [g1(I’), …, gk(I’)]`

Direct (Flow) Dependence in Loops

We say that SA→SB (SAδSB) iff there exist $I, I' \in L$ and $I \leq I'$ where

1. There is a feasible path from instance I of statement SA to instance I' of statement SB,

```
SA:   X[f1(I), …, fk(I)] = ...
      ...
SB:   ... = X[g1(I'), …, gk(I')]
```

2. $f_1(I) = g_1(I'),\ f_2(I) = g_2(I'), \ldots,\ f_k(I) = g_k(I')$

The statement SA in iteration I writes and SB in iteration I' reads from the same memory location M

Antidependence in Loops

We say that SA$\not\rightarrow$SB (SAδ^{-1}SB) iff there exist $I, I' \in L$ and $I \leq I'$:

1. There is a feasible path from instance I of statement SA to instance I' of statement SB,

```
SA:    ... = X[f1(I), …, fk(I))
       ...
SB:    X[g1(I’), …, gk(I’)] = ...
```

2. $f_1(I) = g_1(I'), f_2(I) = g_2(I'), ..., f_k(I) = g_k(I')$

The statement SA in iteration I reads and SB in iteration I' writes to the same memory location M

Output Dependence in Loops

We say that SA↛SB (SAδ^0SB) iff there exist $I, I' \in L$ and $I \leq I'$:

1. There is a feasible path from instance I of statement SA to instance I' of statement SB,

```
SA: X[f1(I), …, fk(I)] = ...
       ...
SB: X[g1(I’), …, gk(I’)] = ...
```

2. $f_1(I) = g_1(I'), f_2(I) = g_2(I'), ..., f_k(I) = g_k(I')$

The statement SA in iteration I and SB in iteration I' both write to the same memory location M

Dependence Testing

Dependence testing requires finding a solution to

$$f_1(I) = g_1(I'),$$
$$f_2(I) = g_2(I'), \ldots,$$
$$f_k(I) = g_k(I')$$

under the inequality constraints $I \in L$ and $I' \in L$

```
do i1 = L_1 to U_1
   do i2 = L_2 to U_2
     . . .
      do ik = L_k to U_k
        statements
      enddo
     . . .
   enddo
enddo
```

Complexity: undecidable in general

- Indirection arrays (e.g. X[Y[i]]). They may only be known at runtime, without a specific application knowledge
- General alias analysis
- Non-linear subscript expressions

Dependence Testing: Formulate

Since we assume affine subscript expressions, each f(I) and g(I) is

$$c_0 + c_1 i_1 + \ldots + c_n i_n,$$

where $i_1 \ldots i_n$ are loop index variables and c's are constants.

So we now have a system of equations

$$a_{10} + a_{11} i_1 + \ldots + a_{1n} i_n = b_{10} + b_{11} j_1 + \ldots + b_{1n} j_n$$

$$\ldots$$

$$a_{k0} + a_{k1} i_1 + \ldots + a_{kn} i_n = b_{k0} + b_{k1} j_1 + \ldots + b_{kn} j_n$$

And for all I: $L_1 \leq i_1 \leq U_1 \ldots L_n \leq i_n \leq U_n$ and same for I'

Instance of integer programming

⇒ NP-complete in general (but don't be scared by it!!!)

Simplifications

Two major simplifications in practice:

- Subscript expressions are usually simple:
  most often i_k or $a_1 i_k + a_0$

- Induction variable transformations help

- Be **conservative**:
  Check if a dependence **may exist**.

Simplifications

ZIV, SIV, MIV A subscript expression containing **z**ero, **s**ingle, or **m**ultiple index variable respectively:

E.g., A[3], A[2 * i1 - 3], A[2 * i1 + 3 * i2 + 5]

Separable Subscripts : A subscript position is said to be **separable** if the index variables used in that subscript position are not used in any other subscript position.

E.g., A[i+1, j, k] and A[i, j, k]

Coupled Subscripts : Two subscript positions are said to be **coupled** if the same index variable is used in both positions.

E.g., A[i+1, i, k] and A[i, j+i, k]

Exact Solutions for SIV

A pair of subscripts with index variable i_k are **Strong SIV** if the subscript expressions are the form $a\ i_k + b_1$ and $a\ i_k + b_2$

- The loop iterates between one and n_k.
- Assumes: n_k, a, b_1, b_2 are known

Dependence exists *iff* either of these hold:

1. $a = 0$ and $b_1 = b_2$, or
2. $|\ d_k\ | \leq n_k - 1$, where $d_k = (b1-b2)/a$ and integer

Proof: We assume I < I' and solve for f(I) = g(I'). Then $a\ I + b_1 = a\ I' + b_2$. We get $I'-I = (b_1 - b_2)/a$. The dependence exists if the formula can be satisfied within the range of the indices, i.e. if this expression is smaller than the loop bound

Some special cases of SIV

Special cases:

- **Weak-zero SIV:** compare $a_1\ i_k + b_1$ with b_2
  $l = (b_2-b_1)/a_1$ and solution exists if $l \leq\ n_k-1$

- **Weak-crossing SIV:** compare $a_1\ i_k + b_1$ with $-a_1\ i_k + b_2$
  Here the distance changes to $(b_2-b_1)/2a_1$

Weak SIV: GCD Test

Simplifications

1. ignore loop bounds!
2. only test if a solution is *possible* (GCD property)
3. test each subscript position separately

GCD Property for Single Variable

Let $f(I) = a_1 I + a_0$ and $g(I) = b_1 I + b_0$

$f(I) = g(I') \Rightarrow a_1 I + a_0 = b_1 I' + b_0$.

GCD Property: If there is a solution to the previous equation, then $g = \gcd(a_1, b_1)$ divides $a_0 - b_0$.

Proof: Let $a_1 = n_1 g$, $b_1 = m_1 g$. Then $g \times (n_1 I - m_1 I') = a_0 - b_0$, and the term in parenthesis must be an integer.

GCD Test for Multiple Indices

Let $f(\mathbf{I}) = a_n i_n + \ldots + a_1 i_1 + a_0$ and
$g(\mathbf{I}) = b_n i_n + \ldots + b_1 i_1 + b_0$.

GCD Property: If there is a solution to the equation

$a_n i_{n1} + \ldots + a_0 = b_n i_{n2} + \ldots + b_0$, then

$g = \gcd(a_1, \ldots, a_n, b_1, \ldots, b_n)$ divides $(a_0 - b_0)$.

More tests: E.g., Banerjee test, Lamport test, Delta test…

Exact Solutions for Weak SIV

The set of subscripts with index variable i_k are **Weak SIV** if the subscripts are of the form $a_1\ i_k + b_1$ and $a_2\ i_k + b_2$

Each such subscript position j gives an equation of the form:

$$a_1\ i_k = a_2\ i_k + b_2 - b_1$$

Approach for each index variable i_k :

1. Solve up to r simultaneous equations in 2 unknowns.
2. Check if solutions satisfy 2 inequalities from the previous slide

Exact Solutions for Weak SIV

Special case: one of a_1 or a_2 is zero: **Weak-Zero SIV** (solution is similar to strong SIV)

General problem: Find if $a_1 I + a_0 = b_1 I' + b_0$

(Lemma) An extended GCD property:
For any pair of values (x, y), the Euclidian GCD algorithm can also compute a triplet (g, n_x, n_y) such that

$$g = n_x x + n_y y = \gcd(x, y)$$

Exact Solutions for Weak SIV

Theorem. Let (g, n_a, n_b) be such a triplet for pair $(a_1, -b_1)$.
Let x_k and y_k be given by:

$$
\begin{aligned}
x_k &= n_a \left(\frac{b_0 - a_0}{g} \right) + k\frac{b_1}{g} \\
y_k &= n_b \left(\frac{b_0 - a_0}{g} \right) + k\frac{a_1}{g}
\end{aligned}
$$

Then (x_k, y_k) is a solution of $a_1 i_1 + a_0 = b_1 i_2 + b_0$ for an integral value of k. Furthermore, for any solution (x,y) there is a k such that $x = x_k$ and $y = y_k$

Solution strategy:

1. Compute x_0, y_0 using the above equations
2. Then find all values of k for which $x_0 + k\, b_1/g$ falls within loop bounds, and similarly for y_k.
3. For dependence to exist, the solution (x_k, y_k) must be within the region bounded by loop bounds

Solving Complicated Indices

E.g. `A[x+2y-1, 2y, z, w+z, v, 1].`

(reminder:)

Separable Subscripts : A subscript position is said to be **separable** if the index variables used in that subscript position are not used in any other subscript position.
E.g., A[i+1, j, k] and A[i, j, k]

Coupled Subscripts : Two subscript positions are said to be **coupled** if the same index variable is used in both positions.
E.g., A[i+1, i, k] and A[i, j+i, k]

Solving Complicated Indices

E.g. `A[x+2y-1, 2y, z, w+z, v, 1].`

Simplify the problem by identifying common cases:

1. Separate subscript positions into coupled groups
2. Label each subscript as ZIV, SIV, or MIV
3. For each separable subscript, apply appropriate test (ZIV, SIV, or MIV).
4. For each coupled group, apply a coupled subscript test; e.g., GCD test or Delta test
5. ***If no test yields independence, a dependence exists:*** Concatenate direction vectors from different groups

6. [10 points]: We studied several tests for independence (ZIV, SIV, MIV, GCD). Which test would you use to test for a possible dependence in the following loops? Apply the test of your choice and report if there are dependences. Assume that the array boundaries are correct. (Use the space to the right for work).

-
```
for (i=0; i<100; i++)
  for (j=0; j<100; i++)
    b[i]=b[i-1]+a[j];
```

-
```
for (i=0; i<n; i++)
  for (j=0; j<n; i++)
     b[3*i-2] = b[2*i+5]+a[j];
```

-
```
for (i=0; i<n; i++)
  for (j=0; j<n; i++)
     b[6*i+2*j+2] = b[2*i+4*j+4]+a[i];
```

Dependence Distance

Dependence Distance: If there is a dependence from statement S1 on iteration I to statement S2 on iteration I' then the corresponding dependence distance vector is

$$d_{I,I'} = [I'_1 - I_1, \ldots I'_k - I_k]$$

Note: Computing distance vectors is harder than testing dependence

Dependence Distance

Direction Vector: For a distance vector of the form $d_{I,I'} = [I'_1 - I_1, \ldots, I'_k - I_k]$ the corresponding direction vector is $\delta_{I,I'} = [\delta_1, \ldots, \delta_k, \ldots \delta_m]$, where

$$\delta_k = \begin{cases} -, & if\ I'_k - I_k < 0 \\ +, & if I'_k - I_k > 0 \\ =, & if I'_k - I_k = 0 \\ *, & if \text{ sign } +, -, = \end{cases}$$

Note: **I** < **J** iff the leftmost non-'=' entry in δ(**I**, **J**) is '+'.

- We use the property of lexicographical ordering

Loop-Independent Dependence

Statement S2 has a loop independent dependence on statement S1 iff S1 references location M on iteration **I**, S2 references M on iteration **I'** and **d(I,I')=0** .

```
do i = 1 to N
    A(i+1) = B(i)
    B(i+1) = A(i+1)
enddo
```

Determines the order in which the code is executed within the nest of loops (compare to loop carried dependence!)

Loop-Carried **Dependence**

Statement S2 has a loop carried dependence on statement S1 iff S1 references location M on iteration **I**, S2 references M on iteration **I'** and **d(I,I')>0** .

```
do i = 1 to N
   A(i+1) = B(i)
   B(i+1) = A(i)
enddo
```

Level of loop-carried dependence is the leftmost non-"=" sign in the direction vector

- Forward dependence: S1 appears before S2 in the loop body
- Backward dependence: S2 appears before S1 in the loop body

Loop-Carried Dependence

Recall: Statement S2 has a loop carried dependence on statement S1 iff S1 references location M on iteration **I**, S2 references M on iteration **I'** and **d(I,I')>0** .

So, in the direction vector for any dependence, the leftmost non-'=' entry must be '+' (if any non-'=' entry is present).

Equivalently: the distance vector $d(I,J) \geq 0$.

Dependence in Loop Nests

```
do v1 = 1 to n1
    do v2 = 1 to n2
      . . .
        do vk = 1 to nk
            X[f1(I), …, fk(I)] = ...
            ... = X[g1(I’), …, gk(I’)]
        end
      . . .
    end
end
```

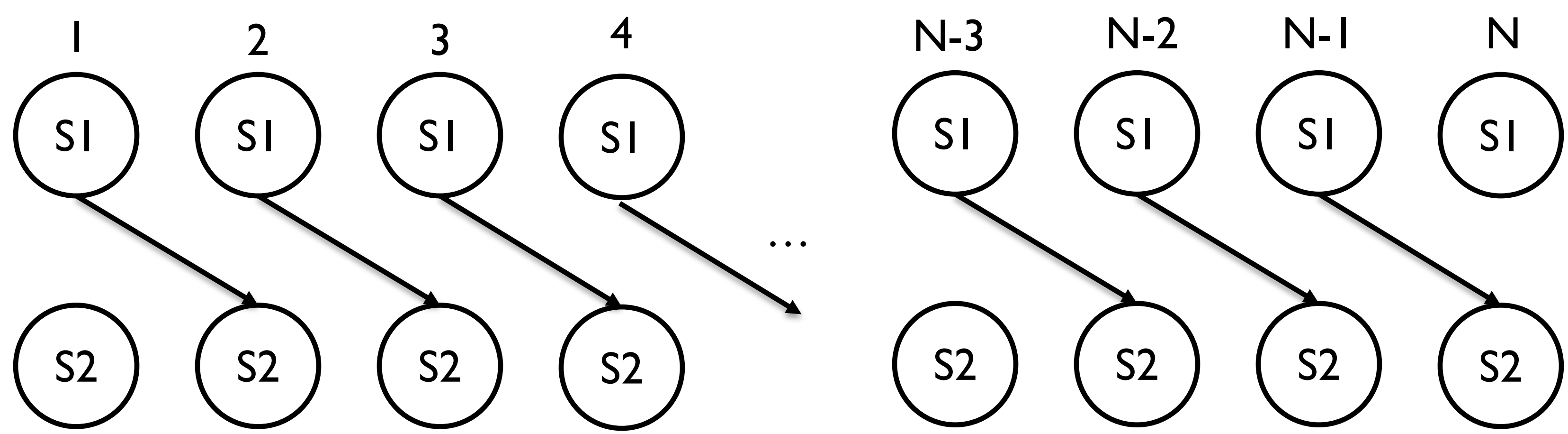
```
I=[v1,v2, …,vk]
I’=[v1’,v2’,…, vk’]
```

A data dependence exists iff ∃I,I’ ∈ [1..n1]x…x [1..nk]
s.t. [f1(I), …, fk(I)] = [g1(I’), …, gk(I’)]

Dependence in Loops

```
int X[], Y[], a[], i;
do i = 1 to N
S1:     X[i+1] = a[i] + 2
S2:     Y[i] = X[i] + 1
enddo
```

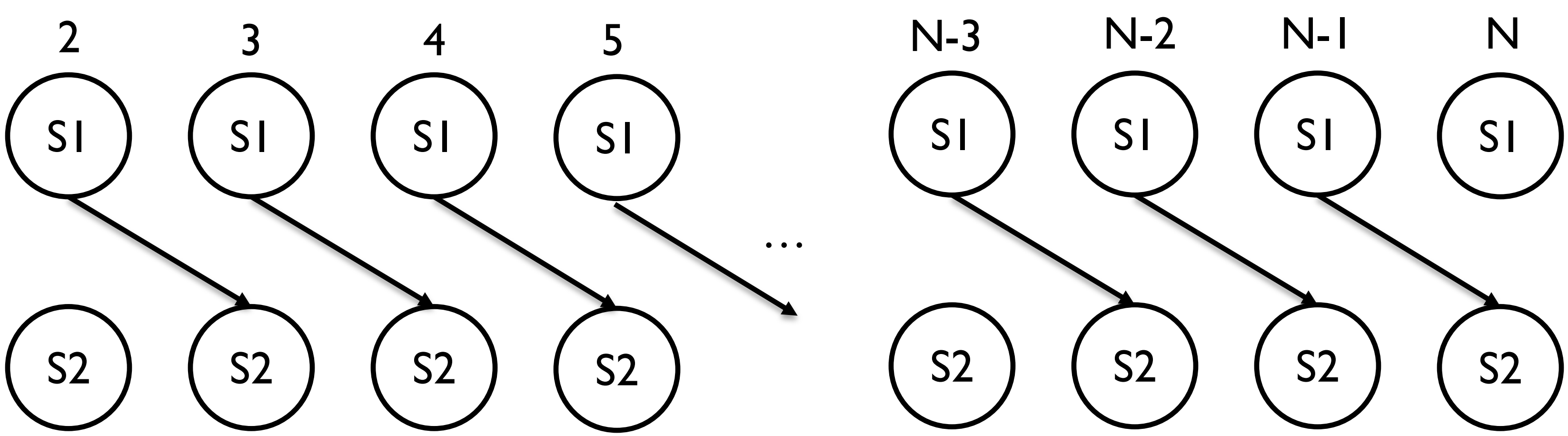
S1->S2
We want:
 d = I' – I
We know:
 I=[i0], I'=[i0']
 f:=i+1, g:=i
Dependence exists if: f(I)=g(I')
---
i0+1=i0'
i0'-i0=1
d=[i0']-[i0]=[1]



Dependence in Loops

```
int X[], Y[], a[], i;
do i = 2 to N
S1:     X[i] = a[i] + 2
S2:     Y[i] = X[i-1] + 1
enddo
```

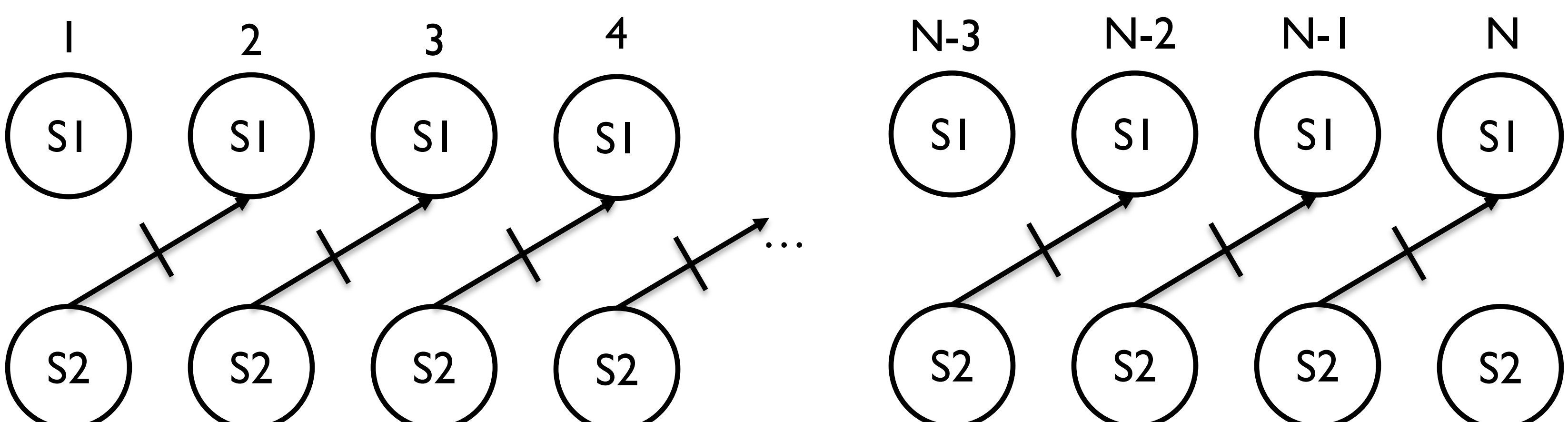
S1->S2
We want:
 d = I' – I
We know:
 I=[i0], I'=[i0']
 f:=i, g:=i-1
Dependence exists if: f(I)=g(I')
---
i0=i0'-1
i0'-i0=1
d=[i0']-[i0]=[1]



Dependence in Loops

```
int X[], Y[], a[], i;
do i = 1 to N
S1:    X[i] = a[i] + 2
S2:    Y[i] = X[i+1] + 1
enddo
```

S2 -|-> S1
We want:
d = I' – I
We know:
I=[i0], I'=[i0']
f:=i+1, g:=i
Dependence exists if: f(I)=g(I')
---
i0+1=i0'
i0'-i0=1
d=[i0']-[i0]=[1]



Dependence in Loops (Examples)

Task: Compute the dependence distance vector

```
      do i = 1 to 100
S1:      X(2*i-1) = X(i) + 1
      enddo
```

```
      do i = 1 to 100
S1:      X(i+1) = X(i/2) + 1
      enddo
```

Dependence in Loops (Examples)

Task: Compute the dependence distance vector

```
      do j = 1 to 10
        do i = 1 to 100
S1:       X(i,j) = W(i,j) + 1
S2:       Y(i,j) = X(100-i,j)
      enddo
```

Dependence in Loops (Examples)

Task: Compute the dependence distance vector

```
    for i = 1 to N
      for j = 1 to M
        for k = 1 to 100
S1:         X(i,j,k+1) = X(i,j,k) + 1
        endfor
      endfor
    endfor
```

Dependence in Loops (Examples)

Task: Compute the dependence distance vector

```
    for i = 1 to N
      for j = 1 to M
        for k = 1 to 100
S1:         X(i+5,j-2,k+1) = X(i-1,j+1,k) + 1
        endfor
      endfor
    endfor
```

Dependence in Loops (Examples)

Task: Compute the dependence distance vector

```
    for i = 1 to N
      for j = 1 to M
        for k = 1 to 100
S1:        X(i,j,k+1) = Y(i,j,k) + 1
S2:        Y(i-1,j+1,k) = X(i+1,j-2,2*k)
        endfor
      endfor
    endfor
```

Control-Flow Analysis

Consider now a program with conditionals:

```
for j = 1 to n {
  A[j] = A[j] * C[j]   // S1
  if (A[j] > k)
     B[j] = B[j] + D[j]   // S2
  else
     B[j] = B[j] - 1.0f
}
```

Control flow dependency exists between S1 and S2
(B[j] will be assigned the value only if A[j] has some value)

Control-Flow Analysis

We can convert the control dependency into a data dependency.
Key steps:

- Consider ***guarded statements*** (if (bool_var) Stmt) and
- Transform the program to ***extract*** complicated expressions from the conditionals

```
for j = 1 to n {
  A[j] = A[j] * C[j]   // S1
  m = A[j] > k
  if (m) B[j] = B[j] + D[j]
  if (!m) B[j] = B[j] – 1.0f
}
```

Control-Flow Analysis (Forward)

```
for j = 1 to n {
  A[j] = A[j] * C[j]   // S1
  m = A[j] > k
  if (m) B[j] = B[j] + D[j]
  if (!m) B[j] = B[j] - 1.0f
}
```

The transformed program preserves all dependencies

This code can be readily vectorized:

- Compute the mask vector m[1…n]
- Compute the then branch result by filtering on m
- Compute the else branch result by filtering on m

E.g., SSE has operations that admit the mask.

Control-Flow Analysis (Exit)

```
for j = 1 to n {
  A[j] = A[j] * C[j]
  if (A[j] > k) break;
  B[j] = B[j] + D[j]
}
```

This is harder to transform with guarded form:

- If the condition is true once, exiting the loop is the same as if it fully executed
- The condition depends on all iterations so far.
- Sketch of a solution. What is missing?

```
for j = 1 to n {
  if (m) break;
  A[j] = A[j] * C[j]
  m = m || A[j] > k
  if (m) break; // ?
  B[j] = B[j] + D[j]

}
```

```
for j = 1 to n {
  m1 = m2
  if (!m1) A[j] = A[j] * C[j]
  if (!m1) m2 = m2 || A[j] > k
  if (!m2) B[j] = B[j] + D[j]
}
```

Control-Flow Analysis (Backward)

```
for j = 1 to n {
      if (A[j] < k) continue;
  S1: k = k + 1
      A[j] = B[k] + D[j]
      if (A[j] > k) goto S1;
}
```

Appears when there is an inner loop like structure

- Applying just the forward analysis would yield potentially wrong code when combined with forward analysis
- It is transformed in conjunction with the related forward branches
- Simple heuristic: identify all code affected by a backward branch untouched and treat as a black-box. However, inefficient; for a more powerful analysis see e.g., *Conversion of Control Dependence to Data Dependence; J.R. Allen and Ken Kennedy; POPL 1983*