CS 526
Advanced Compiler Construction
http://misailo.cs.Illinois.edu/courses/cs526
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**

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

*Slide from Maria Garzaran and David Padua*
Motivation: Vectorization

```c
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];
    }
}
```

```c
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=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 $S_1$ to statement $S_2$ exists if

1. there is a feasible execution path from $S_1$ to $S_2$, and
2. an instance of $S_1$ references the same memory location as an instance of $S_2$ in some execution of the program, and
3. at least one of the references is a store.
Kinds of Data Dependence

**Direct Dependence**

\[
X = \ldots \\
\ldots = X + \ldots
\]

**Anti-dependence**

\[
\ldots = X \\
X = \ldots
\]

**Output Dependence**

\[
X = \ldots \\
X = \ldots
\]
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

**Direct Dependence**

S1: \( X = \ldots \)  
S2: \( \ldots = X + \ldots \)

**Anti-dependence**

S1: \( \ldots = X \)  
S2: \( X = \ldots \)

**Output Dependence**

S1: \( X = \ldots \)  
S2: \( X = \ldots \)
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

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

[Diagram of dependence]
Dependence in Loops

```c
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

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

```c
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
enddo
```

**Rectangular Loop Nest:** The value of n1 to nk does not change during the execution.
Dependence in Loop Nests

**Iteration space**

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

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

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

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

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

Inequalities:
0 ≤ i
i + 0 ⋅ j + 0 ≥ 0
i ≤ 5
−i + 0 ⋅ j + 5 ≥ 0
i ≤ j
−i + j + 0 ≥ 0
j ≤ 7
0 ⋅ i − j + 7 ≥ 0

Turn the inequalities in the form $\alpha \cdot i + \beta \cdot j + \gamma \geq 0$

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

$$
\begin{bmatrix}
1 & 0 \\
-1 & 0 \\
-1 & 1 \\
0 & -1
\end{bmatrix} [i] +
\begin{bmatrix}
0 \\
5 \\
0 \\
7
\end{bmatrix} \geq
\begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}
$$
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 \(n \times n\) matrix
• \(b\) is a vector with length \(n\)
• \(0\) is a vector of \(n\) zeros

Represent a convex polyhedron

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

\(\{i \in \mathbb{Z} \mid \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ n \end{bmatrix} + b \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}\}\)
Dependence in Loop Nests

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

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

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 $[1,2,3] < [1,3,1]$
A data dependence exists iff there exist $I, I' \in [1..n_1] \times \ldots \times [1..n_k]$ such that $[f_1(I), \ldots, f_k(I)] = [g_1(I'), \ldots, g_k(I')]$.
Direct (Flow) Dependence in Loops

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

1. There is a feasible path from instance $I$ of statement $S1$ to instance $I'$ of statement $S2$,

   $SA: \ \ X[f_1(I), \ldots, f_k(I)] = \ldots$
   
   $\ldots$

   $SB: \ \ \ldots = X[g_1(I'), \ldots, g_k(I')]$

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 \( SA \not\leftrightarrow SB \) (\( SA \delta^{-1} SB \)) 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 \( SB \),

\[
SA: \quad \ldots = X[f_1(I), \ldots, f_k(I)] \\
\ldots \\
SB: \quad X[g_1(I'), \ldots, g_k(I')] = \ldots
\]

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 \( I \) reads and \( SB \) in iteration \( I' \) writes to the same memory location \( M \)
Output Dependence in Loops

We say that $SA \leadsto SB$ ($SA \delta^0 SB$) 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 $SB$,

   $SA: X[f_1(I), \ldots, f_k(I)] = \ldots$
   $\ldots$
   $SB: X[g_1(I'), \ldots, g_k(I')] = \ldots$

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 $I$ 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 \( I' \in L \)

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

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
\[
\begin{align*}
  a_{10} + a_{11} i_1 + \ldots + a_{1n} i_n &= b_{10} + b_{11} j_1 + \ldots + b_{1n} j_n \\
  \quad &\vdots \\
  a_{k0} + a_{k1} i_1 + \ldots + a_{kn} i_n &= b_{k0} + b_{k1} j_1 + \ldots + b_{kn} j_n
\end{align*}
\]
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 zero, single, or multiple index variable respectively:

**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 $i_j$ are \textbf{Strong SIV} if the subscript expressions are the form $a_i j + b_1$ and $a_i j + b_2$

- The loop iterates between one and $n_j$.

Dependence exists \textit{iff} either of these hold:

1. $a = 0$ and $b_1 = b_2$, or

2. $|d_j| \leq n_j - 1$, where $d_j = (b_1 - b_2)/a$

\textit{Assumes:} $n_j$, $a$, $b_1$, $b_2$ are known
Exact Solutions for Weak SIV

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

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

$$a_1 i_j = a_2 i_j + b_2 - b_1$$

Approach for each index variable $i_j$:

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 $a_1i_1 + a_0 = b_1i_2 + b_0$

(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

$$g = n_xx + n_yy = \gcd(x, y)$$
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:

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

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.
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 \( f(i) = a_1i + a_0 \) and \( g(i) = b_1i + b_0 \)
\( f(i_1) = g(i_2) \Rightarrow a_1i_1 + a_0 = b_1i_2 + b_0. \)

**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 \( a_1 = n_1g, b_1 = m_1g. \) Then \( g \times (n_1i_1 - m_1i_2) = a_0 - b_0, \) and the term in parenthesis must be an integer.
GCD Test for Multiple Indices

Let \( f(I) = a_k i_k + \ldots + a_1 i_1 + a_0 \) and 
\( g(I) = b_k i_k + \ldots + b_1 i_1 + b_0. \)

**GCD Property:** If there is a solution to the equation 
\( a_k i_k + \ldots + a_0 = b_k i_k + \ldots + b_0, \) then 
\[ g = \gcd(a_1, \ldots, a_k, b_1, \ldots, b_k) \] divides \( (a_0 - b_0). \)

More tests: E.g., Banerjee test, Lamport test, Delta test...
Solving Complicated Indices

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

Simplify the problem by identifying common special 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). Yields direction vectors.
4. For each coupled group, apply a coupled subscript test; e.g., GCD test or Delta test or …
5. If no test yields independence, a dependence exists.
6. 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 $d_{I,I'} = [I_1' - I_1, ..., I_k' - I_k]$ the corresponding direction vector is $\delta_{I,I'} = [\delta_1, ..., \delta_k, ... \delta_m]$, where

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

Note: $I < J$ iff the leftmost non-’=’ entry in $\delta(I,J)$ is ’+’.

- We use the property of lexicographical ordering
Loop-Carried Dependence

Statement $S_2$ has a loop carried dependence on statement $S_1$ iff $S_1$ references location $M$ on iteration $I$, $S_2$ 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: $S_1$ appears before $S_2$ in the loop body
- Backward dependence: $S_2$ appears before $S_1$ in the loop body
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 \( d(i,i') = 0 \).

\[
\text{do } i = 1 \text{ to } N \\
\quad A(i+1) = B(i) \\
\quad B(i+1) = A(i+1) \\
\text{enddo}
\]

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

Recall: Statement $S_2$ has a loop carried dependence on statement $S_1$ iff $S_1$ references location $M$ on iteration $I$, $S_2$ 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

A data dependence exists iff \( \exists I, I' \in [1..n1] \times \ldots \times [1..nk] \)
s.t. \[ f1(I), \ldots, fk(I) = g1(I’), \ldots, gk(I’) \]
Dependence in Loops

int X[], Y[], a[], i;

\( \text{do } i = 1 \text{ to } N \)

\( \text{S1: } X[i+1] = a[i] + 2 \)

\( \text{S2: } Y[i] = X[i] + 1 \)

\( \text{enddo} \)

\( \text{S1} \rightarrow \text{S2} \)

We want: 
\( d = I' - I \)

We know: 
\( I = [i_0], I' = [i_0'] \)
\( f := i+1, g := i \)

Dependence exists if: \( f(I) = g(I') \)

\( \cdots \)
\( i_0 + 1 = i_0' \)
\( i_0' - i_0 = 1 \)
\( d = [i_0'] - [i_0] = [1] \)
Dependence in Loops

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

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;

\[ \text{do } i = 1 \text{ to } N \]

S1: \[ X[i] = a[i] + 2 \]

S2: \[ Y[i] = X[i+1] + 1 \]

enddo

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*

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

```plaintext
do i = 1 to 100
S1: X(i+1) = X(i/2) + 1
enddo
```
Dependence in Loops (Examples)

Task: Compute the dependence distance vector

\[ \text{do } j = 1 \text{ to } 10 \]
\[ \text{do } i = 1 \text{ to } 100 \]
\[ S1: \quad X(i,j) = W(i,j) + 1 \]
\[ S2: \quad Y(i,j) = X(100-i,j) \]
\[ \text{enddo} \]
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 {
    if (A[j] > k)
    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

```java
for j = 1 to n {
    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 {
    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 {
    if (A[j] > k) break;
}

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;
    m = m || A[j] > k
    if (m) break; // ?
}

for j = 1 to n {
    m1 = m2
    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] < m) continue;
    S1: k = k + 1
    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*