Data Dependence

Data Dependence: There is *data dependence* from statement S_1 to statement S_2 if

1. there is a feasible execution path from S_1 to S_2,
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.

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

- one node per statement, and
- a directed edge from S_1 to S_2 if there is a data dependence between S_1 and S_2 (where the instance of S_2 follows the instance of S_1 in the relevant execution).

Data Dependence (cont'd)

Example

```
S1:    X = a + b
S2:    Y = X + c
```

$S_1 \longrightarrow S_2$

Why is this Important?

- Fundamental analysis step for many transformations

  $\Leftarrow$ Any time you need to reorder two statements
- Examples?

Legality of Transformations

Reordering Transformation: A reordering transformation is one that 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.

Legality of Transformations (cont'd)

Theorem:

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

Note 1: Legal $\Rightarrow$ *preserves the meaning of that program*,
i.e., all externally visible outputs are identical to the original program, and in identical order except for exception conditions, and no new exceptions are introduced.

Note 2: If there are conditional statements, the theorem must include control dependences as well as data dependences. Later . . ..

Classes of Data Dependence

Assume a sequential program

Flow Dependence (or True Dependence)

```
S1:    X = a + b
S2:    Y = X + c
```

$S_1 \longrightarrow S_2$

Anti-Dependence

```
S1:    Y = X + b
S2:    X = a + c
```

$S_1 \dashrightarrow S_2$

Classes of Data Dependence (cont'd)

Output Dependence

```
S1:    X = a + b
S2:    X = c + d
```

$S_1 \circ\!\!\rightarrow S_2$

"Input Dependence"

```
S1:    Y = X + b
S2:    Z = X + c
```

$S_1 \times\!\!\rightarrow S_2$

Implications for code transformation

Q. Which dependence types make it illegal to reorder S_1 and S_2?

Q. Illegal $\Rightarrow$ Impossible?

Dependences in Loops

```
   do i = 1 to N
S1:    X(i+1) = a(i) + b(i)
S2:    Y(i) = X(i) + 1
   enddo
```

```
   do i = 1 to N
S1:    X(i) = a(i) + b(i)
S2:    Y(i) = X(i+1) + 1
   enddo
```

```
   do i = 1 to N
S1:    t = a(i) + b(i)
S2:    Y(i) = t + 1
   enddo
```

Note: The dependence graph is a summary of the "unrolled graph".
Consider first loop above with X(i+1) replaced by X(i)
$\Rightarrow$ Some information is lost in the summary.

Dependences in Loop Nests

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

Canonical Loop Nest

A loop nest is said to be in *canonical* form if both lower bound and step of each loop is +1.

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

Note: nj *may vary with* $i1 \ldots i(j-1)$

Rectangular Loop Nest

A loop nest is *rectangular* if the value of $nj, \forall j$, is constant throughout the execution of the loop nest.

Dependences in Loop Nests (cont'd)

Iteration space

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

$$\mathcal{L} = \{[i1, \ldots, in] : 1 \leq i1 \leq n1 \ \wedge \quad \ldots \quad \wedge \ 1 \leq ik \leq nk\}$$

Lexicographic Order

Let $\vec{I_1} = [i1_1,\ i2_1,\ \ldots,\ ik_1]$,
and $\vec{I_2} = [i1_2,\ i2_2,\ \ldots,\ ik_2]$; $\quad \vec{I_1}, \vec{I_2} \in \mathcal{L}$.
$\vec{I_1} \prec \vec{I_2}$ if and only if $\exists j, 1 \leq j \leq k$, such that:

$$i1_1 = i1_2, \quad \ldots\ ,\ i(j-1)_1 = i(j-1)_2, \text{ and } ij_1 < ij_2.$$

Note: Iteration $\vec{I_1}$ precedes iteration $\vec{I_2}$ *iff* $\vec{I_1} \prec \vec{I_2}$.

Definitions of Dependences in Loop Nests (cont'd)

```
do i1 = 1 to n1                                  * Let Ī = (i1, ..., ik)
  do i2 = 1 to n2
   ...
    do ik = 1 to nk
S1:       X(f1(Ī), ..., fr(Ī)) = ...
S2:       ... = X(g1(Ī), ..., gr(Ī))
    enddo
  ...
 enddo
enddo
```

S1 and S2 may be the same statement
or, S1 may occur after S2 in the loop

Definitions of Dependences in Loop Nests (cont'd)

Flow dependence

We say that $S_1 \longrightarrow S_2$ *iff* $\exists\ \vec{I_1}, \vec{I_2} \in \mathcal{L}$, such that

- there is a feasible path from instance $\vec{I_1}$ of S_1 to instance $\vec{I_2}$ of S_2, and
- $f_s(\vec{I_1}) = g_s(\vec{I_2}),\ \forall\ 1 \leq s \leq r$

Anti-dependence

We say that $S_2 \longmapsto S_1$ *iff* $\exists\ \vec{I_1}, \vec{I_2} \in \mathcal{L}$, such that

- there is a feasible path from instance $\vec{I_2}$ of S_2 to instance $\vec{I_1}$ of S_1, and
- $f_s(\vec{I_1}) = g_s(\vec{I_2}), \forall 1 \leq s \leq r$

Improving the Dependence Graphs for Loops

Distance vector : A dependence between statements S_1 and S_2 enclosed in k common loops has *distance vector* $\vec{d} = [d_1, \ \ldots, \ d_k]$ if the iteration with index vector $\vec{I} + \vec{d} = [i_1 + d_1, \ \ldots, \ i_k + d_k]$ depends on iteration with index vector $\vec{I}$.
Note: Computing distance vectors is harder than testing dependence

Direction vector : A vector $\vec{\delta} = [\delta_1, \ \ldots, \ \delta_k]$, where

$$\delta_s = \begin{cases} '=' & \text{if } d_s = 0, \\ '<' & \text{if } d_s > 0, \\ '>' & \text{if } d_s < 0, \\ '*' & \text{if } d_s < 0 \text{ and } d_s > 0 \ and \ d_s = 0. \end{cases}$$

Legal Direction Vectors

Legal Direction Vectors for Dependences

Note: Consider two iteration vectors $\vec{I_1}$ and $\vec{I_2}$ and the direction vector between them, $\delta_{\vec{I_1},\vec{I_2}}$. $\vec{I_1} \prec \vec{I_2}$ if and only if the leftmost non-'=' entry in $\delta_{\vec{I_1},\vec{I_2}}$ is '$<$'.

In the direction vector for any dependence, the leftmost non-'=' entry must be '$<$' (if any non-'=' entry is present).

Equivalently: the distance vector $\vec{d} \geq \vec{0}$.

Loop-Carried and Loop-Independent Dependences

Loop-Carried Dependence : A dependence is a *loop-carried dependence iff* its dependence distance vector $\vec{d} > \vec{0}$.
Equivalently: the leftmost non-'=' entry in the dependence distance vector must be '$<$'.

Loop-Independent Dependence : A dependence is *a loop-independent dependence iff* its dependence distance vector $\vec{d} = \vec{0}$.
Equivalently: all distance vector entries must be '$=$'.

Dependence level : The level of a loop-carried dependence is the position of the leftmost non-'=' entry of its direction vector.
The level of a loop-independent dependence is ∞.

Loop-Carried and Loop-Independent Dependences (cont'd)

Theorem: Any transformation that does not change the order of loops, and does not reorder the iterations of the level-k loop preserves all level-k dependences in that loop.

Theorem: Any transformation that reorders the iterations of the level-k loop and makes no other changes is legal if the loop carries no dependences.

Dependence Testing

Dependence testing requires finding a solution to $\{f_s(\vec{I_1}) = g_s(\vec{I_2}), \forall 1 \leq s \leq r\}$ under the inequality constraints $\vec{I_1}, \vec{I_2} \in \mathcal{L}$.

Complexity

- *Undecidable in general*
  Subscripted subscripts or indirection arrays: $X(index(i,j))$
  - Large class of so-called “irregular” applications
  - Scientific equivalent of arrays of pointers
  - Index array is computed at runtime
    $\Rightarrow$ impossible to test without application-specific knowledge
- Non-linear subscript expressions are extremely hard, and rare

Dependence Testing (cont'd)

- *Assume linear subscript expressions*, e.g.,

$$a_0 + a_1 i_1 + \ldots a_m i_m,$$

where $i_1 \ldots i_m$ are loop index variables.
Instance of integer programming
$\Rightarrow$ *NP-complete in general*

Practical dependence testing

Simplifications

Two major simplifications in practice:

1. Subscript expressions are usually simple
   most often: i_1 or $a_1 i_1 + a_0$
2. Be conservative $\Rightarrow$ Check if a dependence *may* exist.

ZIV, SIV, MIV A subscript expression containing zero, one, or more than one index variable respectively
E.g., `A[n], A[ 2 * i1 + n ], 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, i, 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.

Partitioning the Problem

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 (does not give direction vectors), Delta test
5. *If no test yields independence, a dependence exists*: Concatenate direction vectors from different groups

GCD Test: A Simple Conservative 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) = $a_1 i + a_0$, g(i) = $b_1 i + b_0$
f(i_1) = g(i_2) $\Rightarrow a_1 i_1 + a_0 = b_1 i_2 + b_0$.
GCD Property: If there is a solution to this 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 $g \times (n_1 i_1 - m_1 i_2) = a_0 - b_0$, and the term in parenthesis must be an integer.

GCD Property for Multiple Variables (cont'd)

Let f($\vec{I}$) = $a_k ik + \ \ldots \ + a_0$, g($\vec{I}$) = $b_k ik + \ \ldots \ + b_0$.

GCD Property: If there is a solution to the equation

$a_k ik_1 + \ \ldots \ + a_0 = b_k ik_2 + \ \ldots \ + b_0$, then

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

Exact Solutions for SIV

Assumes: n_j, a, b_1, b_2 *are known*

Strong SIV

A pair of subscripts with index variable i_j are said to be *Strong SIV* if the subscript expressions are the form $ai_j + b_1$ and $ai_j + b_2$.

Dependence exists *iff* either of these hold:

1. a = 0 and $b_1 = b_2$, or
2. $\mid d_j \mid \leq n_j - 1$, where $d_j = \frac{(b_1 - b_2)}{a}$

Exact Solutions for SIV (cont'd)

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:

$$a_1 y = a_2 x + b_2 - b_1$$

Approach for each index variable i_j:

1. Solve up to r simultaneous equations in 2 unknowns.
2. Check if solutions satisfy 2 inequalities (one loop!)

Exact Solution for Separable Weak SIV

The problem

$$a_1 i_1 + a_0 = b_1 i_2 + b_0$$

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

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

Exact Solution for Separable Weak SIV (cont'd)

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:

$$x_k = n_a \left(\frac{b_0 - a_0}{g} \right) + k \frac{b_1}{g}$$

$$y_k = n_b \left(\frac{b_0 - a_0}{g} \right) + k \frac{a_1}{g}$$

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 $y = y_k$.

Strategy

1. Compute x_0, y_0 using the above equations
2. Then find all values of k for which $x_0 + k\frac{b_1}{g}$ falls within loop bounds, and similarly for y_k.