Dependence Analysis and Loop Transformations

CPEG421/621: Compiler Design

University of Delaware

A Quick Overview

We already talked a bit about dependence in the context of instruction scheduling. However dependence analysis provides a more general framework to perform program transformation. It is useful to provide information on:

- How and where to move a given (group of) statement(s)
- How efficient a given transformation will be (i.e. how *profitable* it is)
- If a transformation is legal in general

Dependence analysis is the instrument of choice to perform loop transformations.

Reading List

- The Dragon Book (Chapter 11, esp. Sections 11.3, 11.5, 11.6)
- S. Muchnick's book on advanced compiler technology (Chapter XXX)
- R. Allen's and K. Kennedy's book (esp. Chapters 2,3 for dependence theory; Chapters 5,6 for loop transformations).

This lecture is mainly using the book by Allen & Kennedy.

A Few Definitions

Data Dependence

There is a *data dependence* from statement S_1 to statement S_2, denoted $S_1 \rightarrow S_2$, and which reads “statement S_2 *depends* on statement S_1” if and only if

1. Both statements access the same memory location M, and at least one of them stores into it, and
2. There is a feasible run-time execution path from S_1 to S_2

Types of Dependences (1/2)

They relate closely with read/write dependence used in computer architecture books (such as Hennesy's and Patterson's):

- True (or flow) dependence: $S_1 \delta S_2$ (also written as $S_1 \delta^f S_2$)
- Anti-dependence: $S_1 \delta^{-1} S_2$ (also written as $S_1 \delta^a S_2$)
- Output dependence: $S_1 \delta^o S_2$

Types of Dependences: illustrations (2/2)

True dependence: $S_1 \delta S_2$

```
x                             = /* some source */
/* some destination */ = x
```

Anti-dependence: $S_1 \delta^{-1} S_2$

```
/* some destination */ = x
x                             = /* some source */
```

Output dependence: $S_1 \delta^{o} S_2$

```
x                             = /* some source */
x                             = /* some other source */
```

Dependence in Loops

How do we apply our previous definitions to loops?

Dependence in Loops

How do we apply our previous definitions to loops?

```
for (int i = 0; i < N; ++i) {
    /* S1 */ a[i+1] = a[i] + b[i];
}
```

Dependence in Loops

How do we apply our previous definitions to loops?

```
for (int i = 0; i < N; ++i) {
    /* S1 */ a[i+1] = a[i] + b[i];
}
```

```
for (int i = 0; i < N; ++i) {
    /* S1 */ a[i+2] = a[i] + b[i];
}
```

Dependence in Loops

How do we apply our previous definitions to loops?

```
for (int i = 0; i < N; ++i) {
    /* S1 */ a[i+1] = a[i] + b[i];
}
```

```
for (int i = 0; i < N; ++i) {
    /* S1 */ a[i+2] = a[i] + b[i];
}
```

Some parameterization is necessary to describe loop dependences. Considering *regular* loops, we say a loop always has three components: a lower bound L, an upper bound U, and a step S:

```
for (int i = L; i < S; i += S) {
    /* loop body here */
}
```

Iteration Number and Vector (1/2)

Normalized iteration number

For an arbitrary, *regular* loop in which the loop index I runs from L to S in steps of S, we define the *(normalized) iteration number i* of a specific iteration as : $(I - L + S)/S$, where I is the value of the index on that iteration.

Iteration Vector

Given a nest of n loops, the *iteration vector* $\vec{i}$ of a particular iteration of the innermost loop is defined as: $\vec{i} = [i_1, i_2, \cdots, i_n]$, where n is the innermost loop in the loop nest.

Iteration Number and Vector (2/2)

For an n-level loop: $\vec{i} = \left[i_1, i_2, \cdots, i_n\right]$

```
for (int i1 = L1; i1 < N1; i1 += S1) {
    for (int i2 = L2; i2 < N2; i2 += S2) {
        /* ... Other loops go here */
        for (int in = Ln; i < Nn; in += Sn) {
            /* S */ /* innermost loop body */
        }
    }
}
```

Iteration Number and Vector (2/2)

For an *n*-level loop: $\vec{i} = [i_1, i_2, \cdots, i_n]$

```
for (int i1 = L1; i1 < N1; i1 += S1) {
    for (int i2 = L2; i2 < N2; i2 += S2) {
        /* ... Other loops go here */
        for (int in = Ln; i < Nn; in += Sn) {
            /* S */ /* innermost loop body */
        }
    }
}
```

What does $S[(1,0)]$ represent in the following snippet?

```
for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 2; ++j) {
        S;
    }
}
```

Another Example

```
#include <string.h>
void matvec(int M, int N, double *C, double *A, double *v)
{
    double (*c)[N] = (double (*)[N]) C;
    double (*a)[N] = (double (*)[N]) A;
    memset(C,0,sizeof(double)*M*N);

    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            c[i][j] += a[i][j] * v[j];
}
```

Another Example

```
#include <string.h>
void matvec(int M, int N, double *C, double *A, double *v)
{
    double (*c)[N] = (double (*)[N]) C;
    double (*a)[N] = (double (*)[N]) A;
    memset(C,0,sizeof(double)*M*N);

    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            c[i][j] += a[i][j] * v[j];
}
```

What is the state of the computation in the loop when $\vec{i} = [3, 5]$? When $\vec{i} = [0, 4]$?

Theorem: Loop Dependence

Definition

Iteration $\vec{i}$ *precedes* iteration $\vec{j}$, denoted $\vec{i} < \vec{j}$ if and only if the statements in $\vec{i}$ are all executed before the statements executed in $\vec{j}$. More formally:

$$\vec{i} < \vec{j} \Leftrightarrow \begin{cases} 1. & \vec{i}[1:n-1] < \vec{j}[1:n-1], \mathit{or} \\ 2. & \vec{i}[1:n-1] = \vec{j}[1:n-1] \wedge \vec{i}_n < \vec{j}_n \end{cases} \tag{1}$$

Theorem

In a common n-level loop nest L_n, with two vectors $\vec{i}$, $\vec{j}$ in it, and a common memory location M and δ^* some dependence:

$$\exists \delta^* : S_1 \delta^* S_2 \Leftrightarrow \begin{cases} 1. & \vec{i} < \vec{j} \vee \vec{i} = \vec{j} \wedge \exists path(S_1 \rightarrow S_2) \in L_n \\ 2. & access(S_1 \rightarrow M)_{\vec{i}} \wedge access(S_2 \rightarrow M)_{\vec{j}} \\ 3. & write = access(S_1 \rightarrow M)_{\vec{i}} \vee write = access(S_2 \rightarrow M)_{\vec{j}} \end{cases} \tag{2}$$

Maintaining Program Equivalence

Definition: Equivalence

Two computations are *equivalent* if, on the same input, they produce identical values for output variables at the time output statements are executed and the output statements are executed in the same order.

Definition: Reordering Transformation

A reordering transformation $\mathcal{T}_{\mathcal{R}}$ is any program transformation that merely changes the order of execution of the code, without adding or deleting any execution of any statements

Example of Reordering Transformations

The original code:

```
for (int i = 0; i < N; ++i) {
S1: x = a[i];
    for (int j = 0; j < P; ++j) {
S2:     b[i][j] = x + 3 * b[i][j];
    }
S3: a[i] = b[i][P-1];
}
```

Example of Reordering Transformations

The original code:

```
for (int i = 0; i < N; ++i) {
S1: x = a[i];
    for (int j = 0; j < P; ++j) {
S2:     b[i][j] = x + 3 * b[i][j];
    }
S3: a[i] = b[i][P-1];
}
```

The modified code:

```
for (int i = 0; i < N; ++i) {
S3: a[i] = b[i][P-1];
    for (int j = 0; j < P; ++j) {
S2:     b[i][j] = x + 3 * b[i][j];
    }
S1: x = a[i];
}
```

A reordering transformation can produce incorrect code!

Reordering Transformations

Definition

A reordering transformation $\mathcal{T}_{\mathcal{R}}$ *preserves* a dependence if it preserves the relative execution order of the source and sink of that dependence.

Reordering Transformations

Definition

A reordering transformation $\mathcal{T}_{\mathcal{R}}$ *preserves* a dependence if it preserves the relative execution order of the source and sink of that dependence.

Fundamental Theorem of Dependence

Any reordering transformation $\mathcal{T}_{\mathcal{R}}$ that preserves every dependence in a program preserves the meaning of that program.

Reordering Transformations

Definition

A reordering transformation $\mathcal{T}_\mathcal{R}$ *preserves* a dependence if it preserves the relative execution order of the source and sink of that dependence.

Fundamental Theorem of Dependence

Any reordering transformation $\mathcal{T}_\mathcal{R}$ that preserves every dependence in a program preserves the meaning of that program.

Definition

A transformation is said to be *valid* for the program to which it applies if it preserves all dependences in the program.

A Summarizing Example

```
L0: for (int i = 0; i < N; ++i) {
L1:       for (int j = 0; j < 2; ++j) {
S0:             a[i][j] = a[i][j] + B;
          }
S1:       t       = a[i][0];
S2:       a[i][0] = a[i][1];
S3:       a[i][1] = t;
    }
```

- There is a dependence from S_0 to S_1, S_2 and S_3.
  - → A dependence-based compiler could not accept to reorder loop L_1 with the block $\{S_1, S_2, S_3\}$

A Summarizing Example

```
L0: for (int i = 0; i < N; ++i) {
L1:     for (int j = 0; j < 2; ++j) {
S0:         a[i][j] = a[i][j] + B;
        }
S1:     t       = a[i][0];
S2:     a[i][0] = a[i][1];
S3:     a[i][1] = t;
    }
```

- There is a dependence from S_0 to S_1, S_2 and S_3.
  - $\rightarrow$ A dependence-based compiler could not accept to reorder loop L_1 with the block $\{S_1, S_2, S_3\}$
- However, the interchange leaves the same values in the array `a`.
  - $\rightarrow$ `a[i][0]` and `a[i][1]` receive an identical update (`B`).
- Although there is no dependence between L_1 and block $\{S_1, S_2, S_3\}$ (and thus could be executed in parallel), there is no way to tell it from our current dependence framework.

A Summarizing Example

```
L0: for (int i = 0; i < N; ++i) {
L1:     for (int j = 0; j < 2; ++j) {
S0:         a[i][j] = a[i][j] + B;
        }
S1:     t       = a[i][0];
S2:     a[i][0] = a[i][1];
S3:     a[i][1] = t;
    }
```

- There is a dependence from S_0 to S_1, S_2 and S_3.
  - → A dependence-based compiler could not accept to reorder loop L_1 with the block $\{S_1, S_2, S_3\}$
- However, the interchange leaves the same values in the array `a`.
  - → `a[i][0]` and `a[i][1]` receive an identical update (`B`).
- Although there is no dependence between L_1 and block $\{S_1, S_2, S_3\}$ (and thus could be executed in parallel), there is no way to tell it from our current dependence framework.

→ Our definition of valid transformation is stronger than our definition of computation equivalence.

Distance and Direction Vectors (1/2)

Definition: Distance Vector

Let S_1 on iteration vector $\vec{i}$ and S_2 on iteration vector $\vec{j}$ be two statements so that there is some dependence $S_1 \delta^* S_2$ on the loop nest L_n. Then the dependence distance vector $\vec{d}$ of length n is defined as:

$$\vec{d}_{\vec{i},\vec{j}} = \vec{d}(\vec{i},\vec{j}) = \vec{j} - \vec{i} \tag{3}$$

i.e.

$$d(\vec{i},\vec{j})_k = j_k - j_k, \forall k : 1 \leq k \leq n \tag{4}$$

Distance and Direction Vectors (1/2)

Definition: Distance Vector

Let S_1 on iteration vector $\vec{i}$ and S_2 on iteration vector $\vec{j}$ be two statements so that there is some dependence $S_1 \delta^* S_2$ on the loop nest L_n. Then the dependence distance vector $\vec{d}$ of length n is defined as:

$$\vec{d}_{\vec{i},\vec{j}} = \vec{d}(\vec{i},\vec{j}) = \vec{j} - \vec{i} \tag{3}$$

i.e.

$$d(\vec{i},\vec{j})_k = j_k - j_k, \forall k : 1 \leq k \leq n \tag{4}$$

Definition: Direction Vector

Let S_1 on iteration vector $\vec{i}$ and S_2 on iteration vector $\vec{j}$ be two statements so that there is some dependence $S_1 \delta^* S_2$ on the loop nest L_n. Then the dependence direction vector $\vec{D}$ of length n is defined as:

$$D(\vec{i},\vec{j})_k = \begin{cases} " < " & \text{if } d(\vec{i},\vec{j})_k > 0 \\ " = " & \text{if } d(\vec{i},\vec{j})_k = 0 \\ " > " & \text{if } d(\vec{i},\vec{j})_k < 0 \end{cases} \tag{5}$$

Example for Distance and Direction Vectors

Consider the following code:

```
for (int i = 0; i < N; ++i) {
    for (int j = 0; i < M; ++j) {
        for (int k = 0; i < L; ++k) {
S1:         a[i+1][j][k-1] = a[i][j][k] + 10;
        }
    }
}
```

What are the distance and direction vectors?

Example for Distance and Direction Vectors

Consider the following code:

```
for (int i = 0; i < N; ++i) {
    for (int j = 0; i < M; ++j) {
        for (int k = 0; i < L; ++k) {
S1:         a[i+1][j][k-1] = a[i][j][k] + 10;
        }
    }
}
```

What are the distance and direction vectors?

$\vec{d} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$

$\vec{D} = \begin{bmatrix} < & = & > \end{bmatrix}$

Distance and Direction Vectors (2/2)

Direction Vector Transformation

Let $\mathcal{T}$ be a transformation that is applied to a loop nest and that does not rearrange the statements in the body of the loop. Then the transformation is valid if, after it is applied, none of the direction vectors for dependences with source and sink in the nest has a leftmost non-"$=$" that is "$>$".

What are the distance and direction vectors for the following loop?

```
for (int i = 0; i < 10; ++i) {
    for (int j = 0; j < 100; ++j) {
S1:     a[i][j] = b[i][j]     + X;
S2:     c[i][j] = a[i][99−j] + Y;
    }
}
```

Distance and Direction Vectors (2/2)

Direction Vector Transformation

Let $\mathcal{T}$ be a transformation that is applied to a loop nest and that does not rearrange the statements in the body of the loop. Then the transformation is valid if, after it is applied, none of the direction vectors for dependences with source and sink in the nest has a leftmost non-"$=$" that is "$>$".

What are the distance and direction vectors for the following loop?

```
for (int i = 0; i < 10; ++i) {
    for (int j = 0; j < 100; ++j) {
S1:     a[i][j] = b[i][j]    + X;
S2:     c[i][j] = a[i][99-j] + Y;
    }
}
```

From	`j` = 0 to 48	$\vec{D} = [= \quad <]$
When	`j` = 49	$\vec{D} = [= \quad =]$
From	`j` = 50 to 99	$\vec{D} = [= \quad >]$

Loop-Carried and Loop-Independent Dependences

If we consider two statements, S_1 and S_2, in a loop nest L_n, and assuming there is a dependence $S_1 \delta^* S_2$ over a memory location M, then we have one of:

- Loop-Carried Dependence: S_1 accesses M in one iteration, and S_2 accesses M in a subsequent iteration.
- Loop-Independent Dependence: S_1 and S_2 both access M on the same iteration, but S_1 precedes S_2 during execution of the loop iteration.

Loop-Carried Dependences (1/2)

A simple example:

```
for (int i = 0; i < N; ++i) {
S1:   a[i+1] = f[i];
S2:   f[i+1] = a[i];
}
```

Loop-Carried Dependences (1/2)

A simple example:

```
for (int i = 0; i < N; ++i) {
S1:   a[i+1] = f[i];
S2:    f[i+1] = a[i];
}
```

Definition: Loop-Carried Dependence

Let $\vec{i}$ and $\vec{j}$ be two iteration vectors. Let S_1 and S_2 be two statements. Then $S_1 \delta^* S_2$ is a *loop-carried dependence* on memory location M

$$\iff \begin{cases} 1. & S_1 \textit{ accesses } M \textit{ on } \vec{i} \\ 2. & S_2 \textit{ accesses } M \textit{ on } \vec{j} \\ 3. & \vec{d} > 0 \end{cases} \tag{6}$$

(Remember: $\vec{d} > 0 \Leftrightarrow \vec{D}$ contains a "$<$" as the left-most non-"$=$" component.)

Loop-Carried Dependences (2/2)

Definition: Backward and Forward Dependencies

Let $S_1 \delta^* S_2$ be a loop-carried dependence. It is said to be *backward* if S_2 appears before S_1 in the loop body or if S_1 and S_2 appear in the same statement. If S_2 appears after S_1 in the loop body it is a *forward* dependence.

Definition: Level of a Loop-Carried Dependence

The *level* of a loop-carried dependence is the index of the leftmost non-"$=$" of $\vec{D}$ for the dependence.

Example for Loop-Carried Dependences

What is the direction vector and level of the following loop?

```
for (int i = 0; i < 10; ++i) {
    for (int j = 0; j < 10; ++j) {
        for (int k = 0; k < 10; ++k) {
S1:         a[i][j][k+1] = a[i][j][k];
        }
    }
}
```

Example for Loop-Carried Dependences

What is the direction vector and level of the following loop?

```
for (int i = 0; i < 10; ++i) {
    for (int j = 0; j < 10; ++j) {
        for (int k = 0; k < 10; ++k) {
S1:          a[i][j][k+1] = a[i][j][k];
        }
    }
}
```

$\vec{D} = [= \quad = \quad <]$
Level = 3.

Dependence Preservation

Definition

A dependence is said to be *satisfied* if transformations that fail to preserve it are precluded.
In other words: if we only keep transformations that preserve the initial dependences in a program, then the dependence is satisfied.

Theorem

A reordering transformation $\mathcal{T}_{\mathcal{R}}$ preserves all level-k dependences if and only if

- The iteration order of the level-k loop is preserved
- No loop is interchanged at level $< k$ to a position inside the level-k loop
- Not loop is interchanged at level $> k$ to a position outside the level-k loop

Applying All This to Loops (1/3)

Back to our example:

```
for (int i = 0; i < N; ++i) {
S1:  a[i+1] = f[i];
S2:  f[i+1] = a[i];
}
```

```
for (int i = 0; i < N; ++i) {
S2:  f[i+1] = a[i];
S1:  a[i+1] = f[i];
}
```

Is this transformation (on the right) legal?

Applying All This to Loops (1/3)

Back to our example:

```
for (int i = 0; i < N; ++i) {
S1:  a[i+1] = f[i];
S2:  f[i+1] = a[i];
}
```

```
for (int i = 0; i < N; ++i) {
S2:  f[i+1] = a[i];
S1:  a[i+1] = f[i];
}
```

Is this transformation (on the right) legal? Yes.

Applying All This to Loops (2/3)

A more complicated example. What loops can we legally interchange?

```
for (int i = 0; k < 10; ++i)
  for (int j = 0; j < 10; ++j)
    for (int k = 0; k < 10; ++k)
      a[k+3][j+2][i+1] = a[k][j][i] + B;
```

Applying All This to Loops (2/3)

A more complicated example. What loops can we legally interchange?

```
for (int i = 0; k < 10; ++i)
  for (int j = 0; j < 10; ++j)
    for (int k = 0; k < 10; ++k)
      a[k+3][j+2][i+1] = a[k][j][i] + B;
```

$$\vec{d} = \begin{bmatrix} 3 & 2 & 1 \end{bmatrix}$$

$$\vec{D} = \begin{bmatrix} < & < & < \end{bmatrix}$$

Applying All This to Loops (2/3)

A more complicated example. What loops can we legally interchange?

```
for (int i = 0; k < 10; ++i)
  for (int j = 0; j < 10; ++j)
    for (int k = 0; k < 10; ++k)
      a[k+3][j+2][i+1] = a[k][j][i] + B;
```

$$\vec{d} = \begin{bmatrix} 3 & 2 & 1 \end{bmatrix}$$

$$\vec{D} = \begin{bmatrix} < & < & < \end{bmatrix}$$

```
/* Look at the dependences once we unroll the loop */
for (int i = 0; k < 10; ++i) {
  for (int j = 0; j < 10; ++j) {
    for (int k = 0; k < 10; k += 2 ) {
      a[k+3][j+2][i+1] = a[ k ][ j ][ i ] + B;
      a[k+4][j+3][i+2] = a[k+1][j+1][i+1] + B;
    }
  }
}
```

Applying All This to Loops (3/3)

$$\vec{d} = \begin{bmatrix} 3 & 2 & 1 \end{bmatrix}$$
$$\vec{D} = \begin{bmatrix} < & < & < \end{bmatrix}$$

```
for (int i = 0; k < 10; ++i)
  for (int j = 0; j < 10; ++j)
    for (int k = 0; k < 10; ++k)
      a[k+3][j+2][i+1] = a[k][j][i] + B;
```

Applying All This to Loops (3/3)

$$\vec{d} = \begin{bmatrix} 3 & 2 & 1 \end{bmatrix}$$
$$\vec{D} = \begin{bmatrix} < & < & < \end{bmatrix}$$

```
for (int i = 0; k < 10; ++i)
  for (int j = 0; j < 10; ++j)
    for (int k = 0; k < 10; ++k)
      a[k+3][j+2][i+1] = a[k][j][i] + B;
```

```
for (int i = 0; i < 10; ++i)
  for (int k = 9; k >= 0; k -= 1)
    for (int j = 0; j < 10; j += 1)
      a[k+3][j+2][i+1] = a[k][j][i] + B;
```

Loop-Independent Dependences (1/3)

A dependence is loop-independent if it arises as a result of relative statement position.

Definition: Loop-Independent Dependence

Let $\vec{i}$ and $\vec{j}$ be two iteration vectors. Let S_1 and S_2 be two statements. Then $S_1 \delta^* S_2$ is a *loop-independent dependence* on memory location M

$$\Leftrightarrow \begin{cases} 1. \quad S_1 \; accesses \; M \; on \; \vec{i}, \; S_2 \; accesses \; M \; on \; \vec{j}, \; and \; \vec{i} = \vec{j} \\ 2. \quad \exists \, path(S_1 \rightarrow S_2) \in \vec{i} \end{cases} \tag{7}$$

Loop-Independent Dependences (2/3)

```
for (int i = 0; i < 10; ++i) {
S1:     a[i] = /* some value */
S2:     /* some value */ = a[i];
}

for (int i = 0; i < 10; ++i) {
S1:     a[i] = /* some value */
S2:     /* some value */ = a[9-i];
}
```

Loop-Independent Dependences (3/3)

Theorem

Let S_1 and S_2 two statements. If $S_1 \delta^* S_2$ is loop-independent, any reordering transformation $\mathcal{T}_{\mathcal{R}}$ that does not move statement instances between iterations and preserves the relative order of S_1 and S_2 in the loop body preserves that dependence.

```
   /* Original code */
   for (i = 0; i < N; ++i)
   {
S1:       a[i] = b[i] + C;
S2:       d[i] = a[i] + E;
   }
```

```
   /* Transformed code */
   d[0] = a[0] + E;
   for (i = 1; i < N; ++i)
   {
S1:       a[i-1] = b[i-1] + C;
S2:       d[i]   = a[i]   + E;
   }
   a[n-1] = b[n-1] + C;
```

Is this transformation valid?

Loop-Independent Dependences (3/3)

Theorem

Let S_1 and S_2 two statements. If $S_1 \delta^* S_2$ is loop-independent, any reordering transformation $\mathcal{T}_{\mathcal{R}}$ that does not move statement instances between iterations and preserves the relative order of S_1 and S_2 in the loop body preserves that dependence.

```
   /*  Original  code  */
   for  (i = 0; i < N; ++i)
   {
S1:       a[i] = b[i] + C;
S2:       d[i] = a[i] + E;
   }
```

```
   /*  Transformed  code  */
   d[0] = a[0] + E;
   for  (i = 1; i < N; ++i)
   {
S1:       a[i−1] = b[i−1] + C;
S2:       d[i]   = a[i]   + E;
   }
   a[n−1] = b[n−1] + C;
```

Is this transformation valid? No.

Iteration Reordering

Theorem: Iteration Reordering

A transformation $\mathcal{T}$ that reorders the iterations of a level-k loop, without making any other changes, is valid if the loop carries no dependence.

Simple Dependence Testing

Theorem

Let $\vec{\alpha}$ and $\vec{\beta}$ be iteration vectors within the iteration space of the following loop nest:

```
for (i1 = L1; i1 < U1; i1 += S1) {
    for (i2 = L2; i2 < U2; i2 += S2) {
        /* ... */
        for (in = Ln; in < Un; in += Sn) {
S1:         a[f1(i1,i2,...,in)][f2(i1,i2,...,in)][...][fn(i1,i2,...,in)] = ...;
S2:         ... = a[g1(i1,i2,...,in)][g2(i1,i2,...,in)][...][gn(i1,i2,...,in)];
        }
    }
}
```

$$\exists\, S1\,\delta^*\,S2 \Leftrightarrow \exists\, \vec{\alpha}, \vec{\beta} : \begin{cases} 1. & \vec{\alpha} < \vec{\beta} \text{ lexicographically} \\ 2. & f_i(\vec{\alpha}) = g_i(\vec{\beta})\ \forall i,\ 1 \leq i \leq m \end{cases} \tag{8}$$

Example with a Single Subscript

```
for (i = 0; i < N; ++i) {
S:   a[i+1] = a[i] + B;
}
```

Example with a Single Subscript

```
for (i = 0; i < N; ++i) {
S:  a[i+1] = a[i] + B;
}
```

To test for true dependence on this loop:

- Assume that the left-hand side of statement S accesses memory location M on iteration I_0
- Assume that the right-hand side accesses the same location ΔI iterations later.

$\Rightarrow$ `a[`$I_0 + 1$`]` and `a[`$I_0 + \Delta I$`]` must both refer to the same M.

$$\Rightarrow \; I_0 + 1 = I_0 + \Delta I$$
$$\Rightarrow \; \Delta I = 1$$

If we assume that $N > 0$, and since $\Delta I > 0$ then $\vec{D} = [<]$

Example with Multiple Subscripts

```
for (i = 0; i < 100; ++i)
    for (j = 0; j < 100; ++j)
        for (k = 0; k < 100; ++k)
S:          a[i+1][j][k] = a[i][j][k+1] + B;
```

Example with Multiple Subscripts

```
for (i = 0; i < 100; ++i)
    for (j = 0; j < 100; ++j)
        for (k = 0; k < 100; ++k)
S:              a[i+1][j][k] = a[i][j][k+1] + B;
```

$$\begin{cases} I_0 + 1 = I_0 + \Delta I \\ J_0 = J_0 + \Delta J \\ K_0 = K_0 + \Delta K + 1 \end{cases} \qquad \Longleftrightarrow$$

Example with Multiple Subscripts

```
for (i = 0; i < 100; ++i)
    for (j = 0; j < 100; ++j)
        for (k = 0; k < 100; ++k)
S:          a[i+1][j][k] = a[i][j][k+1] + B;
```

$$\begin{cases} I_0 + 1 = I_0 + \Delta I \\ J_0 = J_0 + \Delta J \\ K_0 = K_0 + \Delta K + 1 \end{cases} \iff \begin{cases} \Delta I = 1 \\ \Delta J = 0 \\ \Delta K = -1 \end{cases}$$

Example with Multiple Subscripts

```
for (i = 0; i < 100; ++i)
    for (j = 0; j < 100; ++j)
        for (k = 0; k < 100; ++k)
S:              a[i+1][j][k] = a[i][j][k+1] + B;
```

$$\begin{cases} I_0 + 1 = I_0 + \Delta I \\ J_0 = J_0 + \Delta J \\ K_0 = K_0 + \Delta K + 1 \end{cases} \iff \begin{cases} \Delta I = 1 \\ \Delta J = 0 \\ \Delta K = -1 \end{cases}$$

$$\vec{D} = [<, =, >]$$

A More Complicated Example

```
for (i = 0; i < 100; ++i)
    for (j = 0; j < 100; ++j)
        a[i+1][j] = a[i][5] + B;
```

A More Complicated Example

```
for (i = 0; i < 100; ++i)
    for (j = 0; j < 100; ++j)
        a[i+1][j] = a[i][5] + B;
```

$$\begin{cases} I_0 + 1 = I_0 + \Delta I \\ J_0 = 5 \end{cases} \iff$$

A More Complicated Example

```
for (i = 0; i < 100; ++i)
    for (j = 0; j < 100; ++j)
        a[i+1][j] = a[i][5] + B;
```

$$\begin{cases} I_0 + 1 = I_0 + \Delta I \\ J_0 = 5 \end{cases} \iff \begin{cases} \Delta I = 1 \\ \Delta J = 5 \end{cases}$$

A More Complicated Example

```
for (i = 0; i < 100; ++i)
    for (j = 0; j < 100; ++j)
        a[i+1][j] = a[i][5] + B;
```

$$\begin{cases} I_0 + 1 = I_0 + \Delta I \\ J_0 = 5 \end{cases} \iff \begin{cases} \Delta I = 1 \\ \Delta J = 5 \end{cases}$$

$$\vec{D} = [<, *]$$

Yet Another Example

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

$$\vec{D}_a = [<, *]$$

Yet Another Example

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

$$\vec{D}_a = [<, *]$$

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

Yet Another Example

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

$$\vec{D}_a = [<, *]$$

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

$$\vec{D}_a = [*, <]$$
$$\vec{D}_a = \{[<, <]\ [=, <]\ [>, <]\}$$

Yet Another Example

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

$$\vec{D}_a = [<, *]$$

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

$$\vec{D}_a = [*, <]$$
$$\vec{D}_a = \{[<, <]\ [=, <]\ [>, <]\}$$

- $[<, <]$ is a level-1 true dependence
- $[=, <]$ is a level-2 true dependence
- $[>, <]$ is a level-1 anti-dependence. It exists because of the following iteration numbers: $j = 0$ and $i = 1$, and $j = 1$ and $i = 0$

Steps to Perform Loop Transformations

1. Assume *affine* iteration space (i.e. something like $f(x) = ax + bx + cx + \cdots + K$ where a, b, c and K are constants)
2. Perform preliminary transformations — See Allen and Kennedy, chapter 4, but you know many of them (they use data flow analysis, def-use chains, etc.):
    - Loop normalization
    - Forward expression substitution
    - Induction-variable substitution (any variable v in a *for* loop L with index i which can be expressed as $v = cstExpr * i + iExpr_L$ where *cstExpr* and *iExpr* do not vary in L)
    - ...
3. Separate the various subscript/index tuples into various groups (not seen here — see Allen and Kennedy, chapter 3 for more complete dependence testing techniques).
4. Asses all the dependences in the loop nest
5. Perform some loop transformation, and test for dependence violation

Loop Interchange

Consider the following loop:

```
for (j = 0; j < N ++j)
    for (i = 0; i < N ++i)
S:      a[i+1][j] = a[i][j] + B;
```

There is a true loop-carried dependence from *S* to itself. But as in C we have a “row-major” type of storing dimensions, the stride used to access `a` is memory-inefficient (and would prevent a vectorizing compiler from vectorizing the loop).

Loop Interchange

Consider the following loop:

```
for (j = 0; j < N ++j)
    for (i = 0; i < N ++i)
S:      a[i+1][j] = a[i][j] + B;
```

There is a true loop-carried dependence from *S* to itself. But as in C we have a “row-major” type of storing dimensions, the stride used to access `a` is memory-inefficient (and would prevent a vectorizing compiler from vectorizing the loop). When interchanging the two loops, we enable multiple optimizing transformations for the compiler:

```
for (i = 0; i < N ++i)
    for (j = 0; j < N ++j)
S:      a[i+1][j] = a[i][j] + B;
```

Safety of Loop Interchange

Not all loops can be interchanged safely. For example:

```
for (j = 0; j < N ++j)
    for (i = 0; i < N ++i)
S:          a[i+1][j] = a[i][j+1] + B;
```

Question: what is the direction vector for this loop?

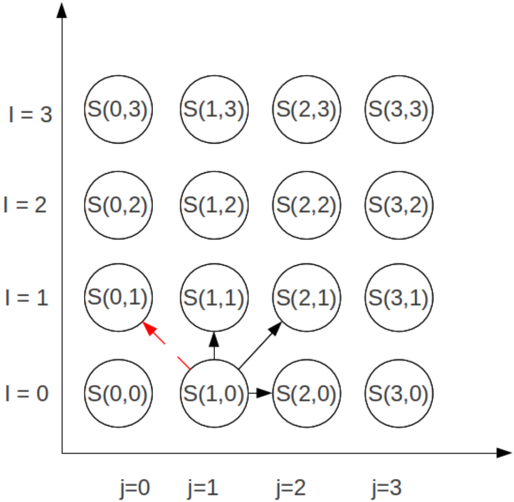
Safety of Loop Interchange

Not all loops can be interchanged safely. For example:

```
for (j = 0; j < N ++j)
    for (i = 0; i < N ++i)
S:      a[i+1][j] = a[i][j+1] + B;
```

Question: what is the direction vector for this loop?
$\vec{D} = [>, <]$
We cannot interchange the two loops!

Illustration of Preceding Example



Safety of Loop Interchange (cont'd)

Definition

A dependence is *interchange preventing* with respect to a given pair of loops if interchanging those loops would reorder the endpoints of the dependence.

Definition

A dependence is *interchange sensitive* if it is carried by the same loop after interchange. That is, an interchange-sensitive dependence moves with its original carrier loop to the new level.

In the previous picture, the dashed red arrow is interchange preventing, while the horizontal and vertical arrows represent interchange-sensitive dependences. The "diagonal" dependence will always be carried by the outermost loop.

Theorem

Let $\vec{D}(\vec{i}, \vec{j})$ be a direction vector for a dependence in a perfect loop nest of n loops. Then the direction vector for the same dependence after a permutation of the loops in the nest is determined by applying the same permutation to the elements of $\vec{D}(\vec{i}, \vec{j})$.

Direction Matrix

Definition

The *direction matrix* for a loop nest L_n is a matrix in which each row is a direction vector for some dependence between statements contained in the next and every such direction vector is represented by a row.

Example:

Its direction matrix is

```
for (k = 0; k < L; ++k)
    for (j = 0; j < M; ++j)
        for (i = 0; i < N; ++i)
        a[i+1][j+1][k] = a[i][j][k] + a[i][j+1][k+1];
```

Direction Matrix

Definition

The *direction matrix* for a loop nest L_n is a matrix in which each row is a direction vector for some dependence between statements contained in the next and every such direction vector is represented by a row.

Example:

```
for (k = 0; k < L; ++k)
    for (j = 0; j < M; ++j)
        for (i = 0; i < N; ++i)
        a[i+1][j+1][k] = a[i][j][k] + a[i][j+1][k+1];
```

Its direction matrix is

$$\begin{bmatrix} < & < & = \\ = & > & < \end{bmatrix}$$

Legality of loop interchange

A permutation of the loops in a perfect nest is legal if and only if the direction matrix, after the same permutation is applied to its columns, has no “$>$” direction as the leftmost non-“=” direction in any row.

Back to Our Example

Its direction matrix is

```
for (k = 0; k < L; ++k)
    for (j = 0; j < M; ++j)
        for (i = 0; i < N; ++i)
        a[i+1][j+1][k] = a[i][j][k] + a[i][j+1][k+1];
```

Back to Our Example

Its direction matrix is

```
for (k = 0; k < L; ++k)
    for (j = 0; j < M; ++j)
        for (i = 0; i < N; ++i)
       a[i+1][j+1][k] = a[i][j][k] + a[i][j+1][k+1];
```

$$\begin{bmatrix} < & < & = \\ < & = & > \end{bmatrix}$$

Is the following loop interchange legal?

Back to Our Example

Its direction matrix is

```
for (k = 0; k < L; ++k)
    for (j = 0; j < M; ++j)
        for (i = 0; i < N; ++i)
        a[i+1][j+1][k] = a[i][j][k] + a[i][j+1][k+1];
```

$$\begin{bmatrix} < & < & = \\ < & = & > \end{bmatrix}$$

Is the following loop interchange legal?

Its direction matrix is

```
for (j = 0; j < M; ++j)
    for (i = 0; i < N; ++i)
        for (k = 0; k < L; ++k)
        a[i+1][j+1][k] = a[i][j][k] + a[i][j+1][k+1];
```

Back to Our Example

Its direction matrix is

```
for (k = 0; k < L; ++k)
    for (j = 0; j < M; ++j)
        for (i = 0; i < N; ++i)
        a[i+1][j+1][k] = a[i][j][k] + a[i][j+1][k+1];
```

$$\begin{bmatrix} < & < & = \\ < & = & > \end{bmatrix}$$

Is the following loop interchange legal?

Its direction matrix is

```
for (j = 0; j < M; ++j)
    for (i = 0; i < N; ++i)
        for (k = 0; k < L; ++k)
        a[i+1][j+1][k] = a[i][j][k] + a[i][j+1][k+1];
```

$$\begin{bmatrix} < & = & < \\ = & > & < \end{bmatrix}$$

Loop Parallelization

Theorem

In a perfect loop nest, a particular loop can be parallelized at the outermost level if and only if the column of the direction matrix for that nest contains only "=" entries.

Example:

```
for (k = 0; k < L; ++k) {
    for (j = 0; j < M; ++j) {
        for (i = 0; i < N; ++i) {
            a[i][j][k+1]
= a[i][j][k] + X1;
            b[i+1][j][k]
= b[i][j][k] + X2;
            c[i+1][j+1][k+1] = c[i][j][k] + X3;
        }
    }
}
```

Its direction matrix is

Loop Parallelization

Theorem

In a perfect loop nest, a particular loop can be parallelized at the outermost level if and only if the column of the direction matrix for that nest contains only "=" entries.

Example:

```
for (k = 0; k < L; ++k) {
    for (j = 0; j < M; ++j) {
        for (i = 0; i < N; ++i) {
            a[i][j][k+1]
= a[i][j][k] + X1;
            b[i+1][j][k]
= b[i][j][k] + X2;
            c[i+1][j+1][k+1] = c[i][j][k] + X3;
        }
    }
}
```

Its direction matrix is

$$\begin{bmatrix} < & = & = \\ = & = & < \\ < & < & < \end{bmatrix}$$

Where can we introduce parallelism?

Loop Parallelization (cont'd)

```
for (k = 0; k < L; ++k) {
    // Parallelize here, for example:
    #pragma parallel omp for default(none) \
                shared (a,b,c,X1,X2,X3) private(i,j)
    for (j = 0; j < M; ++j) {
        for (i = 0; i < N; ++i) {
            a[i][j][k+1]    = a[i][j][k] + X1;
            b[i+1][j][k]    = b[i][j][k] + X2;
            c[i+1][j+1][k+1] = c[i][j][k] + X3;
        }
    }
}
```

Its direction matrix is

Loop Parallelization (cont'd)

```
for (k = 0; k < L; ++k) {
    // Parallelize here, for example:
    #pragma parallel omp for default(none) \
                shared (a,b,c,X1,X2,X3) private(i,j)
    for (j = 0; j < M; ++j) {
        for (i = 0; i < N; ++i) {
            a[i][j][k+1]     = a[i][j][k] + X1;
            b[i+1][j][k]     = b[i][j][k] + X2;
            c[i+1][j+1][k+1] = c[i][j][k] + X3;
        }
    }
}
```

Its direction matrix is

$$\begin{bmatrix} < & = & = \\ = & = & < \\ < & < & < \end{bmatrix}$$