CS 526

Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526

DEPENDENCE ANALYSIS

The slides adapted from Vikram Adve and David Padua

Data Dependence

A data dependence from statement S1 to statement S2 exists if

1. there is a feasible execution path from S1 to S2,
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 + …
```

Antidependence

```
… = X
X  = …
```

Output Dependence

```
X  = …
X  = …
```

Dependence Graph

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).

Kinds of Data Dependence

Direct Dependence

```
X =…
   = X + …
```

$S_1 \longrightarrow S_2$

Antidependence

```
… = X
X  = …
```

$S_1 \mathrel{-\!\!\!\mid\!\!\rightarrow} S_2$

Output Dependence

```
X  = …
X  = …
```

$S_1 \mathrel{-\!\!\ominus\!\!\rightarrow} S_2$

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

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

Note 1: *Legal ⇒ preserves the meaning of that program*,

i.e., **all externally visible outputs are identical to the original program**, and in identical order

Only the exception conditions can appear in a different order, but no new exceptions can be introduced.

Note 2: If there are conditional statements, the theorem must include control dependences as well as data dependences. (We will come back to this soon)

Dependence Graph for Loops

(Repeat) A dependence graph is a graph with:

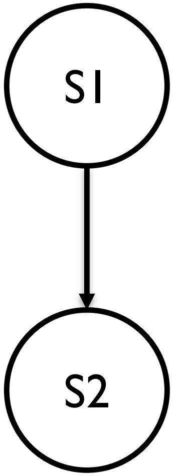
- one node per statement, and
- a directed edge from S1 to S2 if there is a data dependence between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).

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

- Some (detailed) information may be lost

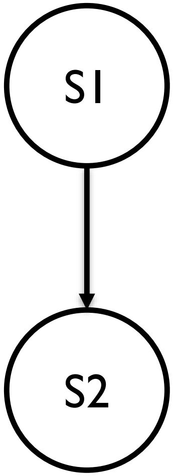
Dependence in Loops

```
        do i = 1 to N
S1:         X(i) = a(i) + 2
S2:         Y(i) = X(i) + 1
        enddo
```



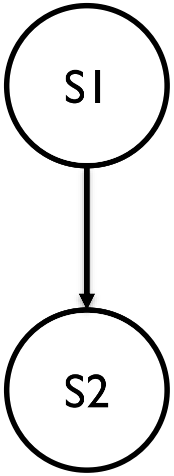
Dependence in Loops

```
        do i = 1 to N
S1:         X(i+1) = a(i) + 2
S2:         Y(i) = X(i) + 1
        enddo
```



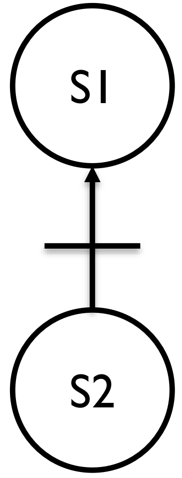
Dependence in Loops

```
        do i = 1 to N
S1:        X(i) = a(i) + 2
S2:        Y(i) = X(i-1) + 1
        enddo
```



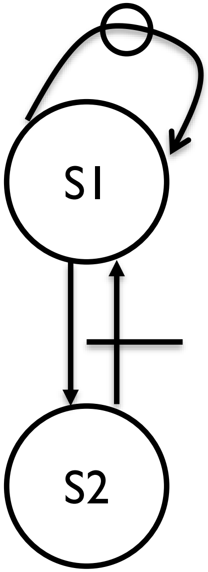
Dependence in Loops

```
        do i = 1 to N
S1:        X(i) = a(i) + 2
S2:        Y(i) = X(i+1) + 1
        enddo
```



Dependence in Loops

```
        do i = 1 to N
S1:         t = a(i) + 2
S2:         Y(i) = t + 1
        enddo
```



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):

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

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

Dependence in Loop Nests

Lexicographic Order: for iteration vectors $[i_1, \ldots, i_n]$ and $[j_1, \ldots, j_n]$:

$[i_1, \ldots, i_n] < [j_1, \ldots, j_n]$ iff there is a subscript k, such that $i_1 = j_1$, ... $i_{k-1} = j_{k-1}$ but $i_k < j_k$

If $\mathrm{I} = [i_1, \ldots, i_n] < [j_1, \ldots, j_n] = J$ we say that the iteration I precedes the iteration *J*

Dependence in Loop Nests

```
do i1 = 1 to n1
   do i2 = 1 to n2
      . . .
         do ik = 1 to nk
            X(f1(I), …, fk(I)) = ...
            ... = X(g1(I), …, gk(I))
         enddo
      . . .
   enddo
enddo
```

I=(i1, i2, …, ik)

Dependence Distance

```
        do i = 1 to N
S1:        X(i) = a(i) + 2
S2:        Y(i) = X(i) + 1
        enddo
```

```
        do i = 1 to N
S1:        X(i+1) = a(i) + 2
S2:        Y(i) = X(i) + 1
        enddo
```

Direct (Flow) Dependence in Loops

We say that S1→S2 iff there exist I, J ∈ L and I ≤ J where

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

```
X(f1(I), …, fk(I)) = ...
    ...
... = X(g1(J), …, gk(J))
```

2. $\mathbf{f_s(I) = g_s(J)}$, $\forall\ 1 \le s \le k$

   The statement S1 in iteration I writes and S2 in iteration J reads from the same memory location M

Antidependence in Loops

We say that S1↛S2 iff there exist I, J ∈ L and I < J where

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

```
          ... = X(f1(I), …, fk(I))
             ...
      X(g1(J), …, gk(J)) = ...
```

2. **$f_s(I) = g_s(J)$**, $\forall\ 1 \le s \le k$

   The statement S1 in iteration I reads and S2 in iteration J writes to the same memory location M

Output Dependence in Loops

We say that S1↬S2 iff there exist I, J ∈ L and I < J where

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

```
X(f1(I), …, fk(I)) = ...
    ...
X(g1(J), …, gk(J)) = ...
```

2. **$f_s(J) = g_s(I)$,** $\forall\ 1 \le s \le k$

   The statement S1 in iteration I and S2 in iteration J both write to the same memory location M

Dependence Distance

Dependence Distance: If there is a dependence from statement S1 on iteration $\vec{i}$ and statement S2 on iteration $\vec{j}$ then the corresponding dependence distance vector is

$$d_{\vec{\boldsymbol{i}},\vec{\boldsymbol{j}}} = [\overrightarrow{j_1} - \overrightarrow{i_1}, \ldots \overrightarrow{j_k} - \overrightarrow{i_k}]$$

Note: Computing distance vectors is harder than testing dependence

Dependence Distance

Direction Vector: For a distance vector of the form $d_{\vec{i},\vec{j}} = [\overrightarrow{j_1} - \overrightarrow{i_1}, \ldots \overrightarrow{j_k} - \overrightarrow{i_k}]$ the corresponding direction vector is $\delta_{\vec{i},\vec{j}} = [\delta_1, \ldots \delta_k]$, where

$$\delta_i = \begin{cases} - , & if\ \overrightarrow{j_1} - \overrightarrow{i_1} < 0 \\ +, & if\ \overrightarrow{j_1} - \overrightarrow{i_1} > 0 \\ =, & if\ \overrightarrow{j_1} - \overrightarrow{i_1} = 0 \\ * , & if\ \text{sign} <,>, = \end{cases}$$

Next time:

Loop independent and loop carried dependencies

Computing dependence vectors vs dependence testing

Uses of Dependency Analysis

Parallelization:

Can we parallelize a particular loop or a loop nest?

Vectorization:

Can we replace scalar instructions (which process single data points) with vector instructions supported by SIMD hardware (which execute one instruction on multiple data points in parallel)?

Uses of Dependency Analysis

```
      do i = 1 to N
S1:      X(i) = a(i) + 2
S2:      Y(i) = X(i) + 1
      enddo
```

```
      do i = 1 to N
S1:      X(i+1) = a(i) + 2
S2:      Y(i) = X(i) + 1
      enddo
```

```
 do i = 1 to N
S1:      t = a(i) + 2
S2:      Y(i) = t + 1
      enddo
```