Dependence analysis

Pattern matching and replacement is all that is needed to apply many source-to-source transformations. For example, pattern matching can be used to determine that the recursion removal transformation presented above is valid.

However, it is often necessary to gather additional information to determine the correctness of a particular transformation.

Dependence information is widely used for this purpose.

A dependence may be due to data or control

Classes of data dependence

Flow dependence (True dependence)

```
S1  X=A+B
S2  C=X+1
```

$$\begin{tikzcd} S1 \arrow[d] \\ S2 \end{tikzcd}$$

Anti-dependence

```
S1  A=X+B
S2  X=C+D
```

$$\begin{tikzcd} S1 \arrow[d, "-" marking] \\ S2 \end{tikzcd}$$

Output dependence

```
S1  X=A+B
      .  .  .
S2  X=C+D
```

$$\begin{tikzcd} S1 \arrow[d, "\circ" marking] \\ S2 \end{tikzcd}$$

The notion of dependence assumes a sequential program.

The equivalent to the notion of dependence for parallel programs has been studies by Shasha and Snir (1988).

Notice that the order of execution of two statements cannot be changed *without further analysis* if one of them is dependent on the other.

Also, they cannot be executed in parallel. This is a stronger condition. Two statements could be interchangeable but not parallelizable.

D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. ACM TOPLAS 10(2). 1988.

Another type of dependence that does not preclude any transformation, but is useful to deal with memory issues is:

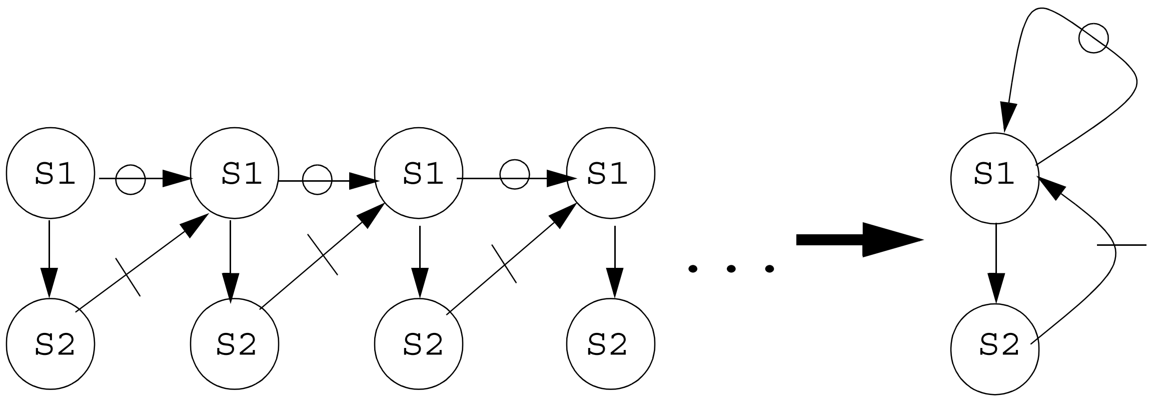
Input-dependence

```
S1 A=X+B
S2 Y=X+D
```

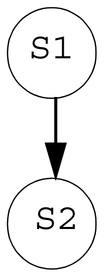
$$\begin{tikzcd} S1 \arrow[d, "\times" marking] \\ S2 \end{tikzcd}$$

Dependences in loops

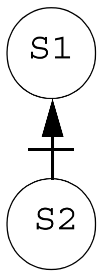
```
   do I=1 to N
S1     A=B(I)+1
S2     C(I)=A+2
```



```
   do I =1 to N
S1     X(I+1)=B(I)+1
S2     A(I)=X(I)
```



```
   do I=1 to N
S1     X(I)=B(I)+1
S2     A(I)=X(I+1)+1
```

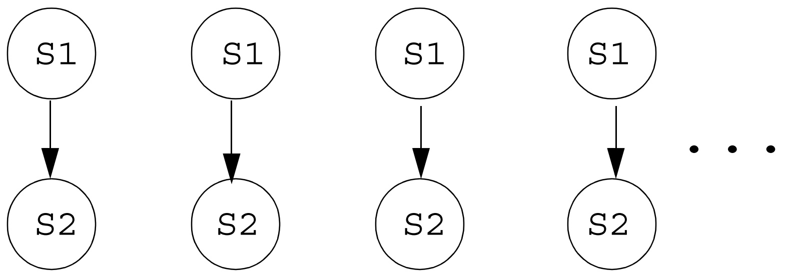


Notice that the dependence graph for a loop is a summary of the “unrolled” dependence graph. Therefore some information is lost.

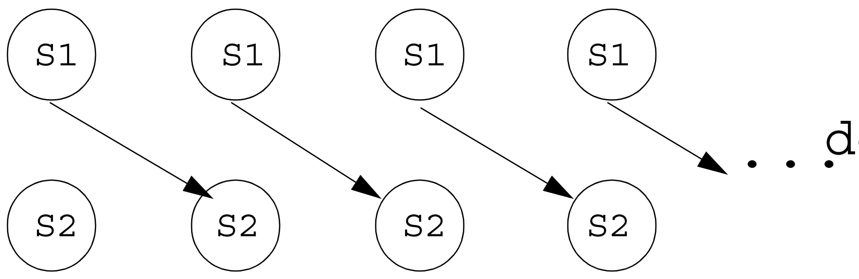
For example, the loop

```
      do I=1 to N
S1       X(I)=B(I)+1
S1       A(I)=X(I)
```

has the same dependence graph as the first loop on the previous page although their unrolled dependence graphs are different as shown on the next page.



```
do ...
   X(I)=
   ...=X(I)
```



```
do ...
   X(I+1) =
   ... = X(I)
```

Definition of dependence in loops

```
      do I=1 to N
S1       X(F(I)) = B(I)+1
S2       A(I) = X(G(I))+2
```

We say that
$$\begin{tikzcd} S_1 \arrow[d] \\ S_2 \end{tikzcd}$$
iff $\exists\ I_1 \leq I_2$ and $I_1, I_2 \in [1,N]$

such that $F(I_1)=G(I_2)$

We say that
$$\begin{tikzcd} S_1 \\ S_2 \arrow[u, "-" marking] \end{tikzcd}$$
iff $\exists\ I_1 < I_2$ and $I_1, I_2 \in [1,N]$

such that $F(I_2)=G(I_1)$

```
      do I=1 to N
S1       A(I) = X(G(I))+1
S2       X(F(I)) = B(I)+2
```

We say that
$$\begin{tikzcd} S_2 \arrow[d] \\ S_1 \end{tikzcd}$$
iff $\exists\ I_1 < I_2$ and $I_1, I_2 \in [1,N]$

such that $F(I_1)=G(I_2)$

Testing the conditions above could be very expensive in general. However, most subscript expressions are quite simple and can be easily analyzed.

The approach that has traditionally been used is to try to *break* a dependence, that is to try to prove that the dependence does not exist.

Practical tests are usually conservative. That is, they may not break a dependence that does not exist. Assuming a dependence that exists is *conservative* but will not lead to incorrect transformations for the cases discussed in this course.

A simple conservative test

The GCD test assumes that

$$F(I) = A_1 I + A_0 \text{ and } G(I) = B_1 I + B_0$$

Then, $F(I_1)=G(I_2)$ iff $A_1 I_1 - B_1 I_2 = A_0 - B_0$

The test breaks the dependence if there is no integer solution to the equation, ignoring the loop limits. This is conservative because the equation could have solutions outside the iteration space only.

There is a solution to the equation $A_1 I_1 - B_1 I_2 = A_0 - B_0$ iff the greatest common divisor of A_1 and B_1 divides $A_0 - B_0$

A more accurate test

To take into account the loop limits we could apply Banerjee's test which proceeds by finding an upper bound U, and a lower bound L of $A_1 I_1 - B_1 I_2$ under the constrains that $1 \leq I_1 \leq I_2 \leq N$.

If either $L > A_0 - B_0$ or $U < A_0 - B_0$, then the functions do not intersect, and therefore we know there is no flow dependence.

For example, consider the loop

```
             do I=1 to 5
        S1      X(I+5) = B(I)+1
        S2      A(I) = X(I)+2
```

If we apply the GCD test, we will find that there is a solution to the equation $A_1 I_1 - B_1 I_2 = A_0 - B_0$ or $I_1 - I_2 = 5$, the dependence will not be broken because the equation has an integer solutions.

However, the upper limit of $A_1 I_1 - B_1 I_2 = I_1 - I_2$ is 4, which is $< A_0 - B_0$ and therefore the dependence would be broken by the second test.

Direction vectors

One way to increase the accuracy of a dependence graph is with direction vectors. For example, the two unrolled dependence graphs on page 8 could be distinguished by annotating the dependence arc with either = (to indicate that the dependence in within the same iteration) or < (to indicate that the dependence goes across iterations).

In general, given a multiply-nested loop

```
do I1=
   do I2=
      ...
         do Id=
            X(F(I1,I2,...,Id))= ...
            ... = X(G(I1,I2,...,Id))
```

our second test would check dependences for each possible *direction.*

For all valid direction vectors $(\Psi_1,\Psi_2,\ldots\Psi_d)$ with each Ψ_i is either <, >, or =, the second test tries to show that there is no solution to the equation:

$$F(I1_1, I2_1, \ldots Id_1) = G(I1_2, I2_2, \ldots Id_2)$$

within the loop limits, with the restrictions:

$$I1_1\ \Psi_1\ I1_2, \quad I2_1\ \Psi_2\ I2_2, \quad \ldots, \quad Id_1\ \Psi_d\ Id_2.$$

These restrictions are taken into account when computing the upper and lower limits (in the complete test, there is a pair of lower and upper limits for each loop index).

If a dependence for a given direction is not broken, an arc annotated with the appropriate direction will be added to the graph.

For more details see (Wolfe and Banerjee 1987).

M. Wolfe and U. Banerjee. Data dependence and its applications to parallel processing. IJPP 16(2). 1987

There are many reasons why Banerjee's test is conservative:

1. It only checks that there is a *real valued* solution to the equations. The solution does not have to be integer. Therefore, failure to break the dependence does not imply that the dependence exists.
2. For multiple subscript arrays, each subscript equation is tested separately. A dependence will be assumed if there is a solution for each separate equation. This is conservative because the system of equations may not have a solution even though each equation has a solution.
3. It is assumed that the loop limits and the coefficients are known constants. This can be relaxed in that the loop limit can be assumed to be infinity and the test would still work. However, if one of the coefficients is not known a dependence is assumed.

Other tests

Pugh (1992) has developed an accurate test that does not have many of the limitations just mentioned.

However, Petersen and Padua (1996) did not detect any significant effect of the Omega test on the parallelism detected in a collection of real codes. However, the Omega test produced a substantially more accurate dependence graph which could be important in many situations.

Blume and Eigenmann (1994) developed a dependence test capable of dealing with symbolic coefficients in subscript expressions.

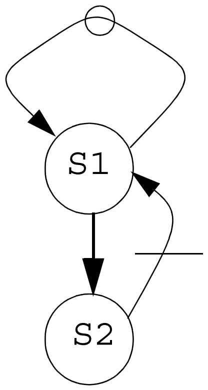
P. Petersen and D. Padua. **Static and Dynamic Evaluation of Data Dependence Analysis Techniques.** IEEE Transactions on Parallel and Distributed Systems. Vol. 7, No. 11, pp. 1121-1132. Nov, 1996. (CSRD Report No. 1509)

W. Blume and R. Eigenman. **The Range Test: A dependence test for symbolic, non-linear expression.** Proceedings of Supercomputing '94. 1994.

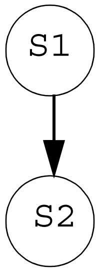
Scalar expansion and privatization

Some arcs in the dependence graph of a loop can be eliminated by using elementary transformations.

```
      DO    I=1,N
S1:         A=B(I)+1
S2:         C(I)=A+D(I)
      END DO
```



```
      DO    I=1,N
S1:         A1(I)=B(I)+1
S2:         C(I)=A1(I)+D(I)
      END DO
      A=A1(N)
```



For a scalar to be expandable, in all iterations the scalar should be written before it is read.

A expandable scalar can also be privatized. That is, when multiple processors cooperate in the execution of a parallel loop, each processor will have its own copy of the private loop variables.

Automatic privatization of scalars is relatively easy. For arrays, the algorithm is substantially more complex. See (Tu and Padua 1973) for more details.

P. Tu and D. Padua. Automatic Array Privatization. Springer-Verlag LNCS 768. 1993.

Control dependences

Informally if a statement Y is control dependent on X then X must have two or more exits. Following one of the exits from X always results in Y being executed, while taking one of the other exits may result in Y not being executed.

X could be an `if` predicate, a computed `goto` statement, a subroutine `call` with multiple locations where to return, etc.

The formal definition, due to Ferrante et al. (1987) is:

Definition. A node Y of a control flow graph is control dependent on node X iff

1. There exists a path from X to Y all of whose interior nodes are post-dominated by Y.
2. X is not post-dominated by Y.

Condition 1 is satisfied by a path consisting of just an edge.

Definition. A node V is postdominated by node W if every path from V to the END node contains W.

J Ferrante et al. The program dependence graph and its use in optimization. ACM TOPLAS 9(3). 1987