CHAPTER 5

Transformations and Vector Code Generation

5.1 Loop Interchange

5.1.1 Introduction

One of the most important parallelism-enhancing transformations is *loop interchange,* which interchanges the order of two perfectly-nested loops. An example of its utility is given by the following code fragment:

```
DO I = 1, N
   DO J = 1, M
      A(I,J+1) = A(I,J) + B
   ENDDO
ENDDO
```

Although the vector code generation algorithm given in Figure 2 on page 153 would produce sequential code, this example can nevertheless be vectorized by interchanging the two loops so that the loop on `I` is innermost.

```
DO J = 1, M
   DO I = 1, N
      A(I,J+1) = A(I,J) + B
   ENDDO
ENDDO
```

Given this modified loop, the vector code generator in Figure 2 now vectorizes in one dimension.

```
DO J = 1, M
   A(1:N,J+1) = A(1:N,J) + B
ENDDO
```

From this example, it is easy to see that loop interchange is also useful for parallelization, because it can be used to bring the parallel loop to the *outermost* position, where it will have larger granularity.

If we think of a loop as creating parameterized instances of the statements within it, we can see that loop interchange is a reordering transformation, because it reorders the execution of the parameterized statements. For example, in the following code, let `S(I,J)` denote the instance of the statement with parameters `I` and `J`.

```
DO J = 1, M
   DO I = 1, N
      S
   ENDDO
ENDDO
```

In the above code `S(1,2)` is executed after `S(2,1)`, but if the loops are interchanged, the order of these two statement instances will be reversed.

Since loop interchange is a reordering transformation, it is governed by the rules of data dependence. That is, it is not legal to make any transformation that reorders the endpoints of any dependence.

In the remainder of this section we will deal with a test for safety of loop interchange, examine the uses of loop interchange and demonstrate its usefulness in a revised algorithm for vector code generation.

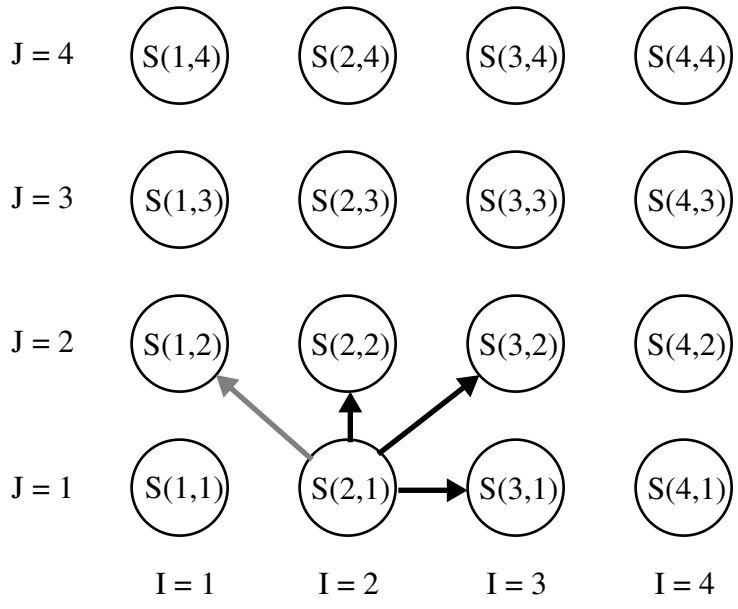
5.1.2 Safety of Loop Interchange

Not all loop interchanges are legal. Consider the following loop.

```
DO J = 1, M
   DO I = 1, N
      A(I,J+1) = A(I+1,J) + B
   ENDDO
ENDDO
```

In the original execution order, `A(2,2)` is assigned when `I=2` and `J=1` and subsequently used one iteration of the `J`-loop later, when `I=1` and `J=2`. If we perform loop interchange, `A(2,2)` is assigned on the second iteration of the outer loop, after being used on the first. Hence, loop interchange would violate the conditions on reordering, since you cannot reorder the endpoints of a dependence. The problem is illustrated by the diagram in

FIGURE 1 Legality of Loop Interchange.



In the diagram, the three black arrows represent dependences that are preserved by loop interchange, while the gray arrow is a dependence that is not preserved.

> **Definition 5.1.** Thus we say that a dependence is *interchange-preventing* if loop interchange would reorder its endpoints. A dependence is *interchange-sensitive* if it is carried by the same loop after interchange.

In the example represented by Figure 1, the horizontal and vertical arrows correspond to interchange-sensitive dependences, while the arrow from S(2,1) to S(3,2) represents a dependence that will always be carried by the outer of the two loops—hence, it is not interchange sensitive.

If we study the interchange-preventing dependence in the above example, we see that it corresponds to direction vector (<,>). Thus, after loop interchange it would have direction vector (>,<), which is clearly illegal. From this it is easy to develop a general test for legality of loop interchange.

> **Definition 5.2.** The *direction matrix* for a nest of loops is a matrix in which each element is a direction and each row corresponds to a dependence between statements contained in the nest and each column corresponds to the loop nesting level, where the loops are numbered from outermost to innermost moving from left to right. In other words, the direction matrix is just a matrix whose rows consist of all the direction vectors of dependences with both endpoints in the nest.

As an example, In the loop nest

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
         A(I+1,J+1,K) = A(I,J,K) + A(I+1,J,K+1)
      ENDDO
   ENDDO
ENDDO
```

The direction matrix is

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

We note that we can derive the direction matrix resulting from any permutation of the loops in a nest by permuting the corresponding columns

of the direction matrix. Hence if we shift the two innermost loops to the outside, moving the old outermost loop to the innermost position, we get the following direction matrix:

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

Note that the new direction vector has a “>” as the leftmost non–“=” symbol. This dependence is illegal, so the transformation is invalid.

> **Theorem 5.1.** 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 column, has no “>” direction as its leftmost non–“=” direction.

Proof. Since a permutation of the loops is simply a combination of loop interchanges, the permutation is a reordering transformation. Hence, it is legal if and only if the order of dependence source and sink is preserved for each dependence in the nest. But if the leftmost non–“=” direction of every dependence is still “<”, then the original source of each dependence is still executed before the original sink. Q.E.D.

Theorem 5.1 gives us a useful test for the legality of a proposed loop permutation.

5.1.3 Effects of Loop Interchange

Recall that the level of a dependence is determined by the position of the leftmost non–“=” direction in the direction vector for that dependence. Loop interchange provides a way to change the level of a dependence, by moving the loop that carries a dependence inward or outward. Let us use the direction matrix to study the effects of loop interchange.

A dependence corresponding to a direction vector that has only one non–“=” direction is always interchange-sensitive—it follows the loop that carries it wherever it is moved. However, dependences with multiple non–“=” directions are more complicated. Consider a dependence with direction vector (<,<,=). If the outermost two loops are interchanged, the dependence will still be carried at the outermost level. On the other hand, if the outermost loop and the innermost loops are exchanged, leaving the middle loop in place, the dependence will be carried at level 2 because the leftmost “<”direction at that level is now the one in the middle.

To see the effect of these principles on code generation for parallelism, consider the following example.

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, L
         A(I+1,J+1,K) = A(I,J,K) + B
      ENDDO
   ENDDO
ENDDO
```

The self-dependence of the single statement in the loop body has direction vector (<,<,=). The innermost loop here is directly vectorizable because it carries no dependence. Hence, on a vector machine, we would generate the following Fortran 90 code:

```
DO I = 1, N
   DO J = 1, M
      A(I+1,J+1,1:L) = A(I,J,1:L) + B
   ENDDO
ENDDO
```

It may not be quite so obvious, however, that we can also vectorize an additional loop. Since the sequential iteration of the outer loop on `I` guarantees that the dependence is satisfied (i.e., the source is always executed before the sink), we can also vectorize the `J`-loop. Because it takes this principle into account, the vectorizer given in Figure 2 on page 153 would produce:

```
DO I = 1, N
   A(I+1,2:M+1,1:L) = A(I,1:M,1:L) + B
ENDDO
```

Since Fortran stores arrays in column-major order, it is usually best to vectorize the loop that iterates over the first dimension. A vectorizer that took this into account, would probably wish to vectorize the `I`-loop instead of the `J`-loop. This can be accomplished by interchanging the two outer loops in the original code. After the interchange, iterating the `J`-loop sequentially satisfies the dependence, making it possible to vectorize both the `I`-loop and the `K`-loop.

```
DO J = 1, M
      A(2:N+1,J+1,1:L) = A(1:N,J,1:L) + B
ENDDO
```

Suppose that we wish to run the original example on a MIMD parallel machine with vector execution units. Since MIMD parallelism usually has significant overhead, our initial goal is to move the parallel loop to

the outermost position. A secondary goal would be to vectorize the loop on I, since it iterates over the dimension that is contiguous in memory. We can do this by swapping the I and K loops, leaving the J-loop in place.

```
DO K = 1, L
   DO J = 1, M
      DO I = 1, N
         A(I+1,J+1,K) = A(I,J,K) + B
      ENDDO
   ENDDO
ENDDO
```

As we saw earlier, this leaves a direction vector (=,<,<), which permits parallelization of the outer loop because it carries no dependence. Iterating the middle loop sequentially satisfies the dependence and makes it possible to vectorize the inner loop:

```
PARALLEL DO K = 1, L
   DO J = 1, M
      A(2:N+1,J+1,K) = A(1:N,J,K) + B
   ENDDO
END PARALLEL DO
```

As we shall see, by examining the direction matrix for a perfect loop nest, we can generate good code for a variety of target architectures.

5.1.4 Loop Interchange and Vectorization

5.1.4.1 Loop Shifting

We now turn to the task of integrating loop interchange into vector code generation. We will begin with a discussion of a simple but effective scheme. Consider the problem of generating code for matrix multiplication. The principal loop nest in a common formulation looks like the following:

```
DO I = 1, N
   DO J = 1, N
      DO K = 1, N
         A(I,J) = A(I,J) + B(I,K)*C(K,J)
      ENDDO
   ENDDO
ENDDO
```

If the vectorizer given in Figure 2 in Chapter 2 is used on this nest, it will produce no vectorization because the dependence is carried by the K-loop and has direction vector (=,=,<). At each stage, starting with the outermost loop, the recursive code generator will discover that the current loop has a recurrence and will generate a sequential do, even though the outer two loops carry no dependence.

This algorithm can do much better if it sequentializes the outermost loop carrying a dependence, rather than the outermost remaining loop. In doing so, it automatically shifts to the inside any loops outside the sequential loop that carry no dependence. In our example, the modified vector code generator would examine the loop at level 1, discover that it is a recurrence but the outermost carried dependence is at level 3. It would then shift the two outer loops inside the K-loop and proceed to vectorize them, since the carried dependence will be satisfied by sequentializing the K-loop:

```
DO K = 1, N
   FORALL J=1,N
      A(1:N,J) = A(1:N,J) + B(1:N,K)*C(K,J)
   END FORALL
ENDDO
```

Here we use a FORALL loop because the two dimensional vector statement is difficult (though not impossible) to write in standard Fortran 90 without it.

The transformation of moving a block of loops inside another loop is called *loop shifting*. Its legality in the case of vector code generation is given by the following Theorem.

> **Theorem 5.2.** In a perfect loop nest, if loops at level i, i+1,...,i+n carry no dependence—that is, all dependences are carried by loops at level less than i or greater than i+n—it is always legal to shift all these loops to the inside of loop i+n+1.

Proof. If loop i+n+1 carries no dependence, then the result is trivial—the transformation can affect no leftmost non–"=" direction. Assume that it carries at least one dependence. Then all dependences it carries have direction vectors with "<" in position i+n+1 and "=" in the first i+n positions. After shifting loops, all these directions will have direction vectors with "<" in position i and "=" in the first i–1 positions. Thus these dependences are still oriented correctly and the transformation is legal. Q.E.D.

A modified multilevel vector code generator based on loop shifting is given in Figure 2.

FIGURE 2 Multi-level vector code generation algorithm with loop shifting.

```
procedure codegen (R, k, D);

    /* R is the region for which we must generate code          */
    /* k is the minimum nesting level of possible parallel loops */
    /* D is the dependence graph among statements in R          */

    find the set {S1, S2, ... , Sm} of maximal strongly-connected
        regions in the dependence graph D restricted to R
        (use Tarjan's algorithm);

    construct Rπ from R by reducing each Si to a single node and
        compute Dπ, the dependence graph naturally induced on
        Rπ by D;

    let {π1, π2, ... , πm} be the m nodes of Rπ numbered in an order
        consistent with Dπ (use topological sort to do the numbering);

    for i = 1 to m do

        if πi is cyclic then

            if the outermost carried dependence is at level p>k then
                shift loops at level k,k+1,...,p–1 inside the level-p loop
            fi

            generate a level-k DO statement;

            let Di be the dependence graph consisting of all
                dependence edges in D that are at level k+1 or greater
                and are internal to πi;

            codegen (πi, k+1, Di);

            generate the level-k CONTINUE statement

        else

            generate a vector statement for πi in ρ(πi)-k+1 dimensions,
                where ρ (πi) is the number of loops containing πi

        fi
    od
end
```

loop-shift code ➜

The correctness of this code generator follows from the correctness of the original algorithm and Theorem 5.2.

5.1.4.2 General Loop Interchange

Experience with the loop shifting algorithm in PFC and the Titan compiler has shown that it is extremely effective in practice. However, it can miss some opportunities for vectorization. Here is an example.

```
DO I = 1, N
   DO J = 1, M
      A(I+1,J+1) = A(I,J) + A(I+1,J)
   ENDDO
ENDDO
```

This loop nest has the direction matrix

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

which foils the loop shifting code generator—the code generated will be sequential at both levels. However, it is easy to see that interchanging the two loops produces vectorization at the innermost level.

```
DO J = 1, M
   A(2:N+1,J+1) = A(1:N,J) + A(2:N+1,J)
ENDDO
```

In the direction matrix, this can be seen by noting that the innermost loop will carry both dependences if exchanged to the outermost position. It should also be noted that there is another good reason to perform this interchange—it causes the innermost loop to iterate over the dimension that is contiguous in memory, which usually makes more efficient use of the memory hierarchy.

How can be fold such sophisticated considerations into the vector code generator? One way is to replace the loop shifting code with a more general heuristic that examines the direction matrix and chooses a loop to iterate sequentially, rearranging the loops in a legal way that makes the desired loop the outermost one.

In a vectorization system, heuristics will often be driven by considerations other than maximum number of vector dimension. On most modern computer systems, the most important factor in performance of the

system, next to parallelism, is performance of the memory hierarchy. Best performance is usually achieved with stride-one access to memory in the innermost loop. Thus, most heuristic loop interchange systems attempt to rearrange the loops to achieve this goal. The following example illustrates how another transformation can be helpful in this process.

```
DO I = 1, N
   DO J = 1, M
      A(I+1,J) = A(I,J+1) + B
   ENDDO
ENDDO
```

In this example, we would like to make the I-loop the innermost loop because that would achieve stride-one access. However, the dependence in this example has the direction vector (<,>), which usually precludes the loop interchange. However, if we reverse the iterations of the J-loop, causing it to count down from M to 1, the loop interchange can take place.

```
DO I = 1, N
   DO J = M, 1, -1
      A(I+1,J) = A(I,J+1) + B
   ENDDO
ENDDO
```

This transformation is called *loop reversal*. To derive the direction vector for a loop after reversal from the direction vector before reversal, simply invert the sense of any non–"=" direction in the position corresponding to the reversed loop. In our example, this yields the direction vector (<,<). Since loop reversal is a reordering transformation, we can see that this particular application is legal because the dependence is carried by a loop outside the reversed loop.

Now that the loop is reversed, it can be exchanged to the outside, and the inner loop can be vectorized.

```
DO J = M, 1, -1
   A(2:N+1,J) = A(1:N,J+1) + B
ENDDO
```

Note that loop reversal did not make more dimensions parallel, rather it helped make the desired loop permutation possible, without sacrificing parallelism.

5.1.4.3 A Loop Selection Heuristic

We are now ready to consider a loop selection heuristic. The basic idea is to replace the loop shifting code with code that effects a more general rearrangement. At level *k*, when a collection of statements forms a recurrence, we will use a greedy heuristic:

1. From the loops at level *k* and deeper, determine which loop would achieve best performance at the innermost position, based on machine characteristics. This loop is called the *desired innermost* loop. Note that the desired innermost loop will usually be the one with the most stride-one access.
2. From among loops that can validly be moved to level *k* using loop interchange and loop reversal, select the loop that would carry the most dependences in that position, so long as it is not the desired innermost loop. Break ties in favor of the outer loop.
3. Select the desired innermost loop if no other choice is feasible or if not selecting it insures that there will be no vectorization.

Let us consider the effect of this heuristic on some examples.First, consider the example on which loop shifting fails:

```
DO I = 1, N
   DO J = 1, M
      A(I+1,J+1) = A(I,J) + A(I+1,J)
   ENDDO
ENDDO
```

If we assume that stride-one access determines the desired innermost loop, then our heuristic would select the J-loop for sequentialization at level 1, which produces the desired code. Note that the J-loop also carries the most dependences. What would have happened if the J-loop had carried no dependences? Then the heuristic selects the I-loop because otherwise there will be no vectorization.

As another example, consider the following loop nest:

```
    DO I = 1, N
       DO J = 1, M
          DO K = 1, L
S1        A(I+1,J+1,K) = B(I,J,K) + C
S2           B(I,J,K+1) = A(I,J,K)* D
          ENDDO
       ENDDO
```

```
      ENDDO
```

The direction matrix for this loop has two rows, one for the dependence from S_1 to S_2 and one for the dependence from S_2 to S_1:

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

Since the loop at level 1 iterates over the first dimension of both matrices, it is the desired innermost loop. Both of the remaining loops in the recurrence would carry one dependence in the outermost position, so we choose the J-loop, because it is the outer of the two. The result can be vectorized in two dimensions, once the order of the two statements is reversed, which must happen to satisfy the second dependence.

```
      DO J = 1, M
S2      B(1:N,J,2:L+1) = A(1:N,J,1:L)* D
S1      A(2:N+1,J+1,1:L) = B(1:N,J,1:L) + C
      ENDDO
```

We now present a general vector code generation algorithm that permits an arbitrary loop selection heuristic. Note that it differs from the loop-shifting version only in the code for selecting the loop to be sequentialized.

FIGURE 3

Multi-level vector code generation algorithm with loop shifting.

procedure *codegen* (*R*, *k*, *D*);

/* *R* is the region for which we must generate code */
/* *k* is the minimum nesting level of possible parallel loops */
/* *D* is the dependence graph among statements in R */

find the set $\{S_1, S_2, \ldots, S_m\}$ of maximal strongly-connected regions in the dependence graph *D* restricted to *R* (use Tarjan's algorithm);

construct R_π from *R* by reducing each S_i to a single node and compute D_π, the dependence graph naturally induced on R_π by *D*;

let $\{\pi_1, \pi_2, \ldots, \pi_m\}$ be the *m* nodes of R_π numbered in an order consistent with D_π (use topological sort to do the numbering);

for i = 1 **to** m **do**

```
            if π_i is cyclic then
                rearrange loops according to a heuristic that optimizes
                    machine performance while insuring that
                    the level-k loop carries at least one dependence;
                generate a level-k DO statement;
                let D_i be the dependence graph consisting of all
                    dependence edges in D that are at level k+1 or greater
                    and are internal to π_i;
                codegen (π_i, k+1, D_i);
                generate the level-k CONTINUE statement
            else
                generate a vector statement for π_i in ρ(π_i)-k+1 dimensions,
                    where ρ (π_i) is the number of loops containing π_i
            fi
        od
    end
```

loop selection →

We now turn to the investigation of transformations that break dependences when vectorization or parallelization fails.

5.2 Recurrence-Breaking Transformations

The existence of a recurrence in a target loop does not always spell disaster for a vector or parallel code generator. As we have seen, loop interchange can often move a loop that does not carry any recurrence into the target position. In addition, there are a number of special transformations that can eliminate various types of recurrences at the cost of a little space or computation time. In this section we will visit a number of these.

Develop the importance of recurrence breaking transformations by presenting examples of code that can be transformed to utilize vector hardware—include scalar expansion (vector swap), pattern recognition (sum reduction), array renaming (may be tough to come up with an example), node splitting (an accumulation into a vector), symbolic resolution (SAXPY). Point out that this chapter will develop methods for handling each of these.

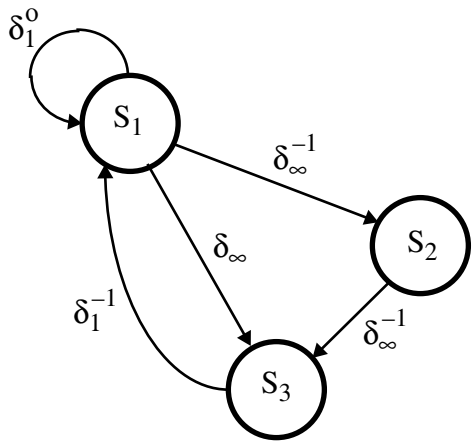
5.2.1 Scalar Expansion

In Fortran 77 programs, scalar variables are frequently used as temporary variables in computations involving vectors and arrays. As an example, consider the following code fragment:

```
    DO I = 1, N
S1     T = A(I)
S2     A(I) = B(I)
S3     B(I) = T
    ENDDO
```

which swaps the contents of two vectors. A standard dependence analyzer, when applied to this example will produce the dependence graph depicted in Figure 4will produce the following dependence graph for this example.

FIGURE 4 Dependence graph for vector swap.



In this graph, the loop independent antidependences are do to arrays `A` and `B`. All other dependences are due to the scalar `T`. In particular, both the carried dependences involve the scalar. The output dependence arises because the same location is used as temporary storage on each iteration. The carried antidependence occurs for the same reason. If we can somehow use a different location on each iteration, both these dependences would disappear and the entire calculation could be vectorized.

To do this, we can replace all references to the scalar temporary T by references to a compiler-generated temporary array T$. This transformation, known as *scalar expansion,* produces the following code.

```
    DO I = 1, N
S1     T$(I) = A(I)
S2     A(I) = B(I)
S3     B(I) = T$(I)
    ENDDO
    T = T$(N)
```

Note the use of the assignment to T after the loop to capture the "last" value computed in the loop.

The loop can now be vectorized in the usual way.

```
S1  T$(1:N) = A(1:N)
S2  A(1:N) = B(1:N)
S3  B(1:N) = T$(1:N)
    T = T$(N)
```

Of course scalar expansion does not come for free—the temporary array introduced by this transformation requires a substantial amount of space. Therefore, it should not be used unless it can be shown to increase the amount of vectorization.

The following example is an analogue of the 5-point stencil calculation that occurs in many Fortran 77 program based on finite differences.

```
    DO I = 1, N
       T = T + A(I) + A(I+1)
       A(I) = T
    ENDDO
```

In this case, there is a use of the scalar variable T before the first assignment to it in the loop. The scalar replacement algorithm must therefore use the value from the previous iteration for any such uses. This leads to the following code:

```
    T$(0) = T
    DO I = 1, N
       T$(I) = T$(I-1) + A(I) + A(I+1)
       A(I) = T$(I)
    ENDDO
    T = T$(N)
```

Note the initialization that is now required because of the upwards-exposed use of T in the loop.

There are two points to be made by this example. First, it shows how to handle the case of a scalar value carried into the next iteration of the loop. Second, it provides an example where scalar expansion is not profitable. In this example, the first statement in the loop cannot be vectorized because the value of T$ computed on each iteration depends on the value from the previous iteration. Although the second statement above can be vectorized, the overall running time of the loop would not improved by the combination of the two transformations. It should be noted that the original loop can be improved by breaking the statement in its body into two:

```
DO I = 1, N
   T1 = A(I) + A(I+1)
   T = T + T1
   A(I) = T
ENDDO
```

Now if we use scalar expansion on T1, we get the following code:

```
DO I = 1, N
   T1$(I) = A(I) + A(I+1)
   T = T + T1$(I)
   A(I) = T
ENDDO
```

in which the finalization statement is omitted because T1 is dead after the loop. Now the first statement can be vectorized to produce:

```
T1$(1:N) = A(1:N) + A(2:N+1)
DO I = 1, N
   T = T + T1$(I)
   A(I) = T
ENDDO
```

in which one of the additions is vectorized.

There are two techniques that can be used to ameliorate the storage problems caused by scalar expansion. The first is by applying a transformation known as *forward substitution*. This can be illustrated by another example:

```
DO I = 1, N
```

```
      T = A(I) + A(I+1)
      A(I) = T + B(I)
ENDDO
```

After scalar expansion, dead code elimination and vectorization, this becomes:

```
T$(1:N) = A(1:N) + A(2:N+1)
A(1:N) = T$(1:N) + B(1:N)
```

The entire temporary array can be eliminated by substituting the right-hand side of the first statement into the right-hand side of the second.

```
A(1:N) = A(1:N) + A(2:N+1)+ B(1:N)
```

The second method is related to the first because it simulated the implementation of a vector statement on a vector register machine. Suppose that most of the advantages of vectorization are realized at vector lengths of 64, possibly because 64 is the length of vector registers on the target machine. We can then *strip mine* the loop into vector operations of length 64 as follows:

```
DO I = 1, N, 64
   DO i = 0, 63
      T$(i) = A(I+i) + A(I+i)
      A(I+i) = T$(i) + B(I+i)
   ENDDO
ENDDO
```

where the code above assumes that `N` is evenly divisible by 64. If this is not the case, the code is still easy to generate but more complicated (this topic will be revisited in Chapter 8. By this technique, the size of the temporary needed has been limited to 64 elements, without significantly affecting machine performance on most vector systems. In fact, vector register machines with vector registers of length 64 will use exactly this technique to implement complex vector expressions, allocating the 64-element temporary arrays to vector registers.

A simple algorithm for scalar expansion is given in Figure 30. It takes advantage of the lack of control flow in the loop by assuming that any assignment to the scalar variable under consideration kills the variable value—that is, the value available on entry to a loop can propagate no further than the first assignment. Hence, the references to `T$(I-S)` are used in place of the scalar reference up to the first assignment. Therefter, all references are to `T$(I)`.

FIGURE 5

Scalar Expansion Algorithm.

Input: A loop with no control flow and a recurrence involving a scalar variable.

Output: A transformed version of the loop in which all references to the variable `T` are replaced by array references to the expanded array `T$`.

for each reference to `T` in the loop **do**

  if the reference is a use of `T` occuring before the first assignment to `T` **then**
    replace `T` with `T$(I-S)` where `I` is the loop induction variable and `S` is the step size of the loop
  else
    replace `T` with `T$(I)`
  endif;
endfor;

place the assignment "`T$(L-S) = T;`" in the loop prologue, where "`L`" is the loop lower bound;
place the assignment "`T = T$(`*M*`)`," where *M* is the value of the loop index `I` on loop exit; i.e., "*M* `= L + FLOOR((`*M*`-L)/S)*S + S;`"

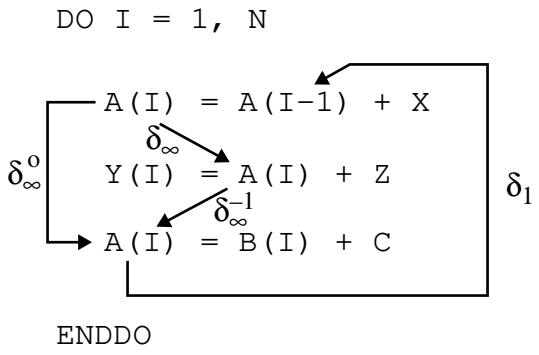
Scalar expansion in the presence of control flow within the loop has special pitfalls. These will be discussed in a future chapter.

5.2.2 Forward Substitution

Briefly develop the conditions under which an expanded scalar can be forward substituted correctly. Point out the advantages of loop carried dependence in this scenario, and point out when substitution can be profitably carried out vs. unprofitable substitution.

5.2.3 Scalar and Array Renaming

In loops that are free of control flow, it is possible to eliminate output dependences. Consider:



The problem with this loop is that the references to A(I) in the first two statements really amount to references to a different variable because `A(I)` is killed by the assignment in the third statement. Thus we could replace `A(I)` by `A$(I)` in those two statements to yield:

```
DO I = 1, N
   A$(I) = A(I-1) + X
   Y(I) = A$(I) + Z
   A(I) = B(I) + C
ENDDO
```

in which both the antidependence and the output dependence in the original loop have been eliminated. This breaks the recurrence and the vectorization algorithm can now produce

```
A(1:N) = B(1:N) + C
A$(1:N) = A(0:N-1) + X
Y(1:N) = A$(1:N) + Z
```

The algorithm for scalar and array renaming is given in

FIGURE 6

Scalar and Array Renaming Algorithm.

Input: A loop with no control flow and one or more output dependences, none of which span more statements than in one iteration of the loop. Note that this restriction can always be satisfied by unrolling the loop.

Output: A transformed version of the loop in which all output dependences have been eliminated by renaming.

for each output dependence δ^o in the loop **do**

let L be the reference on the left hand side of the statement S_0 at the origin of δ^o;
replace L in S_0 with the compiler-generated temporary T;

for each statement S between S_0 and the sink of δ^o **do**
if there is a true dependence from S_0 to reference *R* in S **then**
replace *R* with *T*
endif
endfor;

if δ^o is carried by the loop **then**
let *L′* be the left-hand side of the sink of the dependence;
insert "*T* = *L′* (0)" before the loop, where *L′* (0) denotes *L′* with the loop index replaced by the initial loop index value;
insert "*L*(U) = *T*" after the loop, where *L*(U) denotes *L* with the loop index replaced by the loop upper bound;
endif;

apply scalar expansion to *T*
endfor

The correctness of this algorithm follows from the fact that, if there is no control flow in the loop, the sink of an output dependence kills the output from the source.

We present one final example, to illustrate the use of this algorithm on carried output dependences.

```
DO I = 1, N
   A(I) = A(I) + C
   A(I+1) = A(I) + D
ENDDO
```

There is an output dependence from the second statement in the body to the first and, when taken together with the loop-independent true dependence, it forms a recurrence. Following the algorithm in Figure 7, we get the following code.

```
A$ = A(1)
DO I = 1, N
   A(I) = A$ + C
   A$ = A(I) + D
ENDDO
A(N+1) = A$
```

Scalar expansion of A$ then produces

```
A$(0) = A(1)
DO I = 1, N
   A(I) = A$(I-1) + C
```

```
   A$(I) = A(I) + D
ENDDO
A(N+1) = A$(N)
```

which vectorizes nicely:

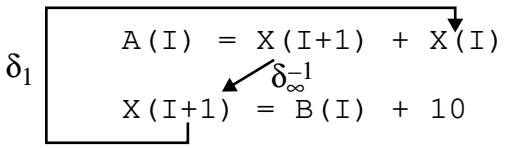
```
A$(0) = A(1)
A(1:N) = A$(0:N-1) + C
A$(1:N) = A(1:N) + D
A(N+1) = A$(N)
```

By showing the effects of array renaming on the graph, develop the two conditions where array renaming can be profitably employed. Briefly describe the changes that should be made to the tree to utilize this.

5.2.4 Node Splitting

Recurrences that involve at least one antidependence may be eliminated through a technique called *node splitting*. This transformation introduces assignments to new array names to eliminate the antidependence in the recurrence. As an example, consider the loop:

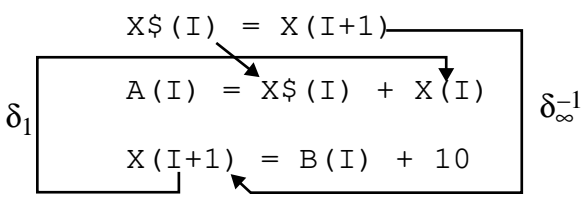
```
DO I = 1, N
```




```
ENDDO
```

The problem in this example is that the first of the statements in the loop body uses both old values (from before the loop) of `X(I+1)` and new values of `X(I)`. If we copy all the old values of `X` into a temporary array `X$` before the first statement, the recurrence is broken:

```
DO I = 1, N
```




```
ENDDO
```

If the standard vectorization algorithm is applied to this code, it will reorder the last two statements and generate the following vector code:

```
X$(1:N) = X(2:N+1)
X(2:N+1) = B(1:N) + 10
A(1:N) = X$(1:N) + X(1:N)
```

The node splitting algorithm is straightforward, and is given in Figure 7.

FIGURE 7

Node Splitting Algorithm.

Input: A recurrence with a single critical antidependence (which would break the cycle if it were broken.

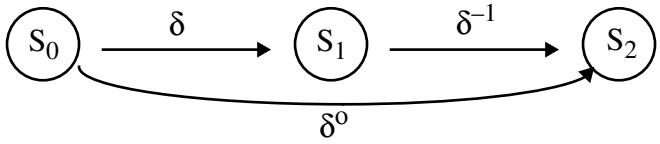
Output: The loop transformed to break the recurrence.

let R be the source of the antidependence;
create a new statement $T = R$, where T is a new compiler-generated temporary, and insert prior to the statement at the origin of the antidependence;
insert the temporary T in place of R in the statement where it originally appeared;
apply scalar expansion to T;

Theorem 5.1. The node-splitting algorithm given in Figure 7, breaks the dependence cycle.

Proof. Recall that the antidependence beginning at R is critical—that is, if it is broken, so is the recurrence. Let S denote the statement originally at the origin of the antidependence and S' the new statement inserted by the algorithm. Clearly, there is no longer an antidependence from S, so that statement can no longer be involved in a recurrence.

The only question that remains is whether the new statement S', which is now the source of the antidependence, can now be part of a recurrence. Suppose that it is. Then there must be a true dependence edge from a third statement S_0 to the reference R. Let R' denote the sink of the original antidependence. The existence of a true dependence from S_0 to R in S and an antidependence from R to R' implies an output dependence from S_0 to R', contradicting the assumption that the original antidependence was critical.



Q.E.D.

We note that the combination of an antidependence and an output dependence such as the one in the proof above cannot occur in a loop on which renaming has been performed.

5.2.5 Index-set Splitting

Even when the dependence pattern in a loop does not permit vectorization, it may be possible to break the loop up into vectorizable portions. In this section, we present a number of *index-set splitting* transformations that improve vectorization in this way.

5.2.5.1 Threshold Analysis

Recall that the *threshold* of a dependence is the value of its leftmost nonzero distance. In essence, the threshold is the number of iterations of the carrier loop crossed by a carried dependence. The important observation about thresholds is this: if the number of iterations in a loop is less than or equal to the threshold of a carried dependence, that dependence can be ignored for the purpose of vectorizing or parallelizing that loop, as illustrated by the following loop:

```
DO I = 1, 20
   A(I+20) = A(I) + B
ENDDO
```

This can be directly rewritten as the vector statement:

```
A(21:40) = A(1:20) + B
```

Even when the number of iterations is far larger than the threshold, the loop can be strip mined to achieve vectorization:

```
DO I = 1, 100
   A(I+20) = A(I) + B
ENDDO
```

After strip mining, the loop becomes:

```
DO I = 1, 100, 20
   DO i = I, I+19
      A(i+20) = A(i) + B
   ENDDO
ENDDO
```

in which the inner loop vectorizes:

```
DO I = 1, 100, 20
   A(I+21:I+40) = A(I:I+19) + B
ENDDO
```

Although this transformation seems promising, it is not very useful in practice because dependences with long thresholds seldom occur in real programs. However, some programs that simulate dynamic storage allocation by assigning different abstract arrays to different sections of the same declared array.

Another kind of threshold that is useful in vectorization is the *crossing threshold*, which occurs in loops where the dependence distance varies. An example is the following loop:

```
DO I = 1, 100
   A(101-I) = A(I) + B
ENDDO
```

Here the dependence of the statement upon itself varies in distance from 99 at `I`=1 down to 1 at `I`=50, after which the references cross. If we split the loop into two at the crossing threshold of 50, we get:

```
DO I = 1, 100, 50
   DO i = I, I+49
      A(101-i) = A(i) + B
   ENDDO
ENDDO
```

in which the inner loop vectorizes

```
DO I = 1, 100, 50
   A(101-I:51-I) = A(I:I+49) + B
ENDDO
```

Crossing Thresholds can be computed in a natural way by the dependence analyzer, as shown in Section 3.4.2.4 on page 82.

Recall what thresholds are, and pull the section out of the tree killer to show how they can be used in general. Temper by saying that this is probably not practical, but look ahead by noting that thresholds may have benefit in other contexts.

5.2.5.2 Loop Peeling

A more frequent occurrence is a loop with a carried dependence whose source is a single iteration.

```
DO I = 1, N
   A(I) = A(I) + A(1)
ENDDO
```

Here, the computation on each iteration uses the value of `A(1)` computed on the first iteration. This carried dependence can be eliminated by *peeling* the first iteration into the loop prologue:

```
A(1) = A(1) + A(1)
DO I = 2, N
   A(I) = A(I) + A(1)
ENDDO
```

The resulting loop carries no dependence and can be directly vectorized:

```
A(1) = A(1) + A(1)
A(2:N) = A(2:N) + A(1)
```

Loop peeling can involve iterations other than the first and last, in which case the loop must be split into two loops, the first ending with the iteration causing the dependence, as shown in this example:

```
DO I = 1, N
   A(I) = A(FLOOR(N/2) + B(I)
ENDDO
```

becomes

```
M = FLOOR(N/2)
DO I = 1, M
   A(I) = A(FLOOR(N/2) + B(I)
ENDDO
DO I = FLOOR(N/2)+1, N
   A(I) = A(FLOOR(N/2) + B(I)
ENDDO
```

This is clearly another form of crossing threshold and the loop size can be easily determined at dependence testing time. In this case, the test that determines the transformation is the *weak-zero test*, described in Section 3.4.2.3 on page 81.

5.2.5.3 Section-based Splitting

A variation on loop peeling is illustrated the following example:

```
DO I = 1, N
   DO J = 1, FLOOR(N/2)
```

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

This loop has a complex dependence pattern—there is a loop-independent dependence from statement S_1 to statement S_2 and a dependence from S_1 to S_2, carried by the I-loop. As a result, no vectorization can be achieved by the standard algorithm. However, notice that S_1 and the loop surrounding it define locations B(1:FLOOR(N/2)), while statement S_2 uses B(1:N). This suggests that we could divide the first loop into two:

```
    DO I = 1, N
       DO J = 1, FLOOR(N/2)
S1        B(J,I) = A(J,I) + C
       ENDDO
       DO J = 1, FLOOR(N/2)
S2        A(J,I+1) = B(J,I) + D
       ENDDO
       DO J = FLOOR(N/2)+1, N
S3        A(J,I+1) = B(J,I) + D
       ENDDO
    ENDDO
```

There is no dependence from S_1 to S_3 so the standard vectorization algorithm will distribute the I-loop and move the loop containing S_3 as follows:

```
    DO I = 1, N
       DO J = FLOOR(N/2)+1, N
S3        A(J,I+1) = B(J,I) + D
       ENDDO
    ENDDO
    DO I = 1, N
       DO J = 1, FLOOR(N/2)
S1        B(J,I) = A(J,I) + C
       ENDDO
       DO J = 1, FLOOR(N/2)
S2        A(J,I+1) = B(J,I) + D
       ENDDO
    ENDDO
```

This permits vectorization of S_3 in two dimensions, while the remainder will vectorize only in the J-dimension:

```
    M = FLOOR(N/2)+1
S3  A(M+2:N+1,2:M+1) = B(M+1:N,1:N) + D
    DO I = 1, N
S1     B(1:M,I) = A(1:M,I) + C
S2     A(1:M,I+1) = B(1:M,I) + D
    ENDDO
```

In order to find this kind of transformation, dependences in loops need to be annotated with the array sections that flow along the dependence edges. Subtraction of such sections led to the loop split in the example shown above.

5.2.5.4 Run-time Symbolic Resolution

In cases where a symbolic variable is unknown, the vectorizer needs to make the most conservative possible assumption. Consider the following loop:

```
DO I= 1, N
   A(I+L) = A(I) + B(I)
ENDDO
```

Because L is unknown, the statement cannot be vectorized. However, it is fairly straightforward for the dependence analyzer to annotate dependence edges with *breaking conditions*, which are Fortran 90 expressions for when a dependence does not exist. In the above example, the dependence has breaking condition (L.LE.0).

When a breaking condition is present, the loop can be vectorized when protected by an IF statement on this condition:

```
IF (L.LE.0) THEN
   A(1:N+L) = A(I) + B(I)
ELSE
   DO I= 1, N
      A(I+L) = A(I) + B(I)
   ENDDO
ENDIF
```

Breaking conditions are particularly useful with computations with variable strides:

```
DO I = 1, N
```

```
      A(I*IS-IS+1) = A(I*IS-IS+1) + B(I)
   ENDDO
```

This type of loop is common in packages like LINPACK [] designed to support a variety of different strides. The loop above can be vectorized if the stride `IS` is anything other than zero. Hence the following code should be generated:

```
M = N*IS-IS+1
IF (IS.NE.0) THEN
   A(1:M:IS) = A(1:M:IS) + B(1:N)
ELSE
   A(1) = A(1) + SUM(B(1:N))
ENDIF
```

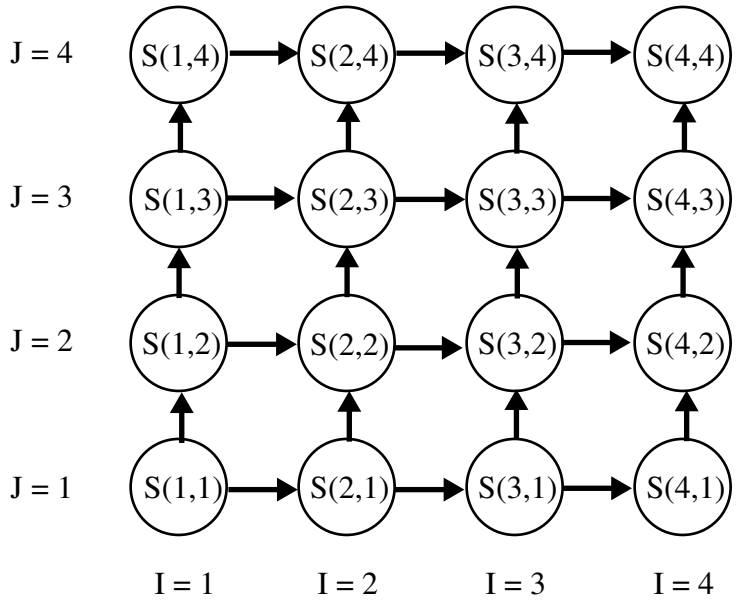
Note that, if the stride is zero, the loop becomes a sum reduction to `A(1)`.

5.2.6 Loop Skewing

The loop skewing transformation can sometimes uncover parallelism where seemingly none exists. Loop skewing is most often used to achieve wavefront-style parallelism. As an example consider the loop

```
   DO I = 1, N
      DO J = 1, N
                        (=,<)
S:       A(I,J) = A(I-1,J) + A(I,J-1)
                   (<,=)
      ENDDO
   ENDDO
```

FIGURE 8 Dependence Pattern before Loop Skewing.



The diagram in Figure 1 shows the dependence pattern for the instances of statements in the double loop nest. The problem is the existence of dependences carried by both the `I`- and `J`-loops.

If we introduce the new index variable `j` and make the variable substitution:

```
j = J + I
```

then we have:

```
J = j - I
```

Substituting this in the original example loop, we get

```
    DO I = 1, N
       DO j = I+1, I+N
                        (=,<)
S:        A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
                 (<,<)
```

```
      ENDDO
   ENDDO
```

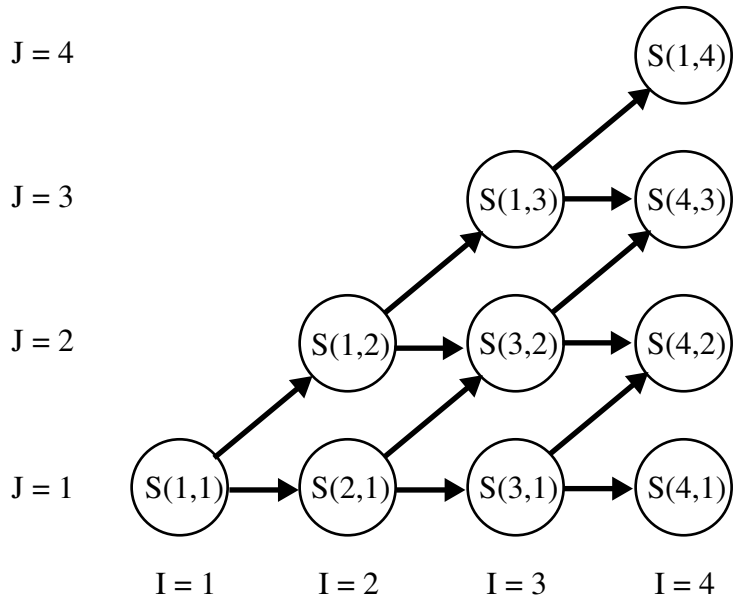
Note the new direction vector for the lower dependence. It is now (<,<) because the value of `I` at the source of the dependence must be less than the value at the sink of the dependence. In fact, examining the first subscript position tells us that the difference must be 1. Since `I` is now subtracted from the index `j` of the second loop, the difference in `j` at source and sink must also be 1. Using delta notation:

$$\Delta I = 1 \text{ and } j - I = j + \Delta j - I - \Delta I$$

$$\text{therefore, } \Delta j = \Delta I = 1.$$

The dependence pattern for the new loop is depicted in Figure 1.

FIGURE 9 Dependence Pattern after Loop Skewing.



Because the second dependence is now interchange-sensitive, indicated by a diagonal orientation in the Figure, the different iterations in the `J`-dimension can now be run independently. If we interchange the loops in the example, both dependences will be carried by the outer loop because the dependence formerly carried by the J-loop is not interchange sensi-

tive—it will still be carried by the outer loop after interchange. The transformed code looks like this:

```
DO j = 2, N+N
   DO I = max(1,j-N), min(N,j-1)
      A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
   ENDDO
ENDDO
```

Note the complex bounds that result from transforming the trapezoidal loop generated by loop skewing. Vector code can be generated as follows:

```
DO j = 2, N+N
   FORALL I = max(1,j-N), min(N,j-1)
      A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
   END FORALL
ENDDO
```

The FORALL statement is required because the vector statement is not directly expressible in triplet notation.

Because the bounds of the FORALL statement contain the outer loop index, the vector bounds be recomputed on each iteration of the outer loop, a significant overhead. This is one of the main disadvantages of loop skewing. A second disadvantage is that it reduces average vector length. In the example above, the vector lengths on successive iterations run from 1 up to N and then back down to 1 in steps of 1, for an average vector length of N/2. Depending on the vector start-up time and the value of N, this transformation may or may not be profitable.

5.2.7 Pattern Recognition

Most vector machines have special hardware to compute several different kinds of recurrences. These almost always include sum reduction, product reduction and max/min reduction. As an example, consider sum reduction.

```
S = 0.0
    DO I = 1, N
       S = S + A(I)
    ENDDO
```

If we assume that floating-point addition is commutative and associative, the computation can be rearranged to compute a series of partial sums in

parallel. If the target machine has a four-stage addition pipeline, the sum reduction can be decomposed into four separate sum reduction computations:

```
S = 0.0
DO k = 1, 4
   SUM(K) = 0.0
   DO I = k, N, 4
      SUM(k) = SUM(k) + A(I)
   ENDDO
   S = S + SUM(k)
ENDDO
```

If we apply loop distribution to this computation, we get

```
S = 0.0
DO K = 1, 4
   SUM(K) = 0.0
ENDDO
DO k = 1, 4
   DO I = k, N, 4
      SUM(k) = SUM(k) + A(I)
   ENDDO
ENDDO
DO k = 1, 4
   S = S + SUM(k)
ENDDO
```

If the loops in the middle nest are interchanged, the result is

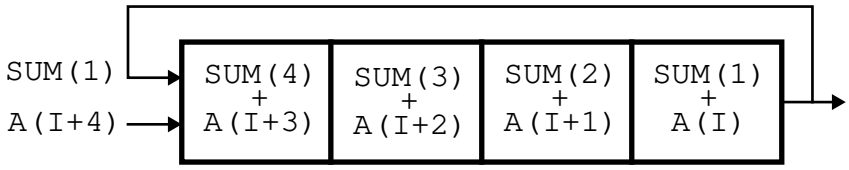
```
DO I = 1, N, 4
   DO k = I, min(I+3,N)
      SUM(k) = SUM(k) + A(I)
   ENDDO
ENDDO
```

Although the inner loop still carries a dependence, it can be vectorized because the threshold of the dependence, namely 3, is less than the maximum number of iterations in the loop (4):

```
DO I = 1, N, 4
   SUM(1:4) = SUM(1:4) + A(I:min(I+3,N))
ENDDO
```

If we note that the vector computation for `I+4` can begin immediately after the vector computation for I has begun, we can completely fill the pipeline, as depicted in Figure 10.

FIGURE 10 Pipeline for Sum Reduction.



Once the four partial sums are available in `SUM(1:4)` the total sum can be computed by three floating point additions, two of which may be overlapped. Thus, sum reduction can be computed at nearly full vector speed. Similar tricks can be used to compute product and min/max reduction as well.

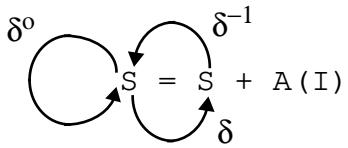
On most machines, the Fortran 90 intrinsic function `SUM` will produce the fastest possible sum reduction that is permitted by commutativity and associativity. Thus, the automatic vectorizer need only replace the original sum reduction loop by

```
S = SUM(A(1:N))
```

The Fortran 90 intrinsic functions `PRODUCT`, `MINVAL` and `MAXVAL` play the same role for multiplication, minimum and maximum, respectively.

So how can a compiler recognize when to use one of these routines. Let us examine the dependence graph for a sum reduction loop.

```
DO I = 1, N
```




```
ENDDO
```

This distinctive pattern can easily be recognized by a compiler—it need only look for a statement with a loop-carried output dependence on itself, combined with a loop-carried true dependence and a loop-carried antidependence.

Sum reduction should be carefully used, as the following example shows.

```
DO I = 1, N
   DO J = 1, M
      S(I) = S(I) + A(I,J)
   ENDDO
ENDDO
```

The inner loop is a sum reduction and, hence, the code could be replaced by

```
DO I = 1, N
      S(I) = S(I) + SUM (A(I,1:M)
ENDDO
```

However, this would not be the best possible as we could also interchange the two loops and vectorize the `I`-loop:

```
DO J = 1, M
   S(1:N) = S(1:N) + A(1:N,J)
ENDDO
```

Develop the ideas behind sum reduction, product reduction, and max-min reduction. Point out how the heuristics developed in scalar expansion very carefully do not interfere with these reductions. Also point out that the transformations must be carefully arranged, by showing an example where sum-reduction may defeat a loop interchange.

5.2.8 Transformations in a Vector Code Generator

Many of the transformations we have described interfere with one another. In Section 5.2.7 on page 176 we saw how premature recognition of sum reduction can interfere with loop interchange. As another example, we show how premature scalar expansion can make recognition of recurrences difficult:

```
DO I = 1, N
   S = S + A(I)
ENDDO
```

We have already seen how this can be reduced to the optimal code by sum reduction recognition. However, a naive vectorizer might try scalar expansion first:

```
S$(0) = S
DO I = 1, N
```

```
      S$(I) = S$(I-1) + A(I)
ENDDO
S = S$(N)
```

The resulting loop is still a recurrence, but is now harder to recognize as a sum reduction. Some systems will replace the above with a call to simple linear recurrence solver, which is better than no vectorization, but less efficient than sum reduction.

From these considerations, it is clear that the transformations must be tried in some order. Once the standard loop-shifting vectorizer has failed to produce vector code for a statement or collection of statements, recurrence breaking transformations should be tried in the following order:

1. pattern recognition
2. scalar expansion
3. renaming to eliminate output dependences
4. node splitting to eliminate antidependence
5. index set splitting
6. loop skewing

These transformations can be incorporated into an algorithm that applies them only in those cases where additional vectorization will be produced.