Data Dependence and Program Restructuring

Michael Wolfe

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-007

May, 1990

Data Dependence and Program Restructuring

Michael Wolfe
Oregon Graduate Institute of Science and Technology
19600 NW von Neumann Drive
Beaverton, OR 97006

mwolfe@cse.ogi.edu
503-690-1153

Abstract
Data dependence concepts are reviewed, concentrating on and extending previous work on direction vectors. A bit vector representation of direction vectors is discussed. Various program restructuring transformations, such as loop circulation (a form of loop interchanging), reversal, skewing, sectioning (strip mining), combing and rotation, are discussed in terms of their effects on the execution of the program, the required dependence tests for legality, and the effects of each transformation on the dependence graph. Simple bit vector operations for the dependence tests and modifying the direction vector are shown. Finally, a simple method to interchange complex convex loop limits is given, which is useful when several loop restructuring operations are being applied in sequence.

Keywords: data dependence, direction vector, restructuring, loop interchanging, loop reversal, loop skewing, loop rotation, sectioning, strip mining

1 Prologue

Analysis of the dependence relations in a program allows the discovery of implicit parallelism in sequential programs, and is used in compilers for today's vector and parallel computers. This analysis can also be used to restructure programs to take advantage of various architectural features, such as local or cache memory and multicomputer topologies.

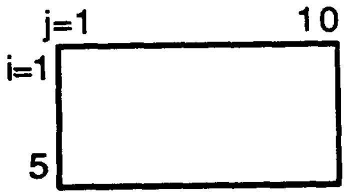
In this paper we quickly review data dependence concepts, concentrating on the data dependence direction vector. We extend direction vectors to distinguish ***reduction operations*** (such as sum, product, dot product, etc.) from other dependence relations. We study the use of a ***bit vector*** representation of the direction vector to implement legality tests for several restructuring transformations, and show how those transformations affect the dependence relations. The transformations discussed are loop interchanging, loop skewing, loop reversal, loop rotation, sectioning (strip mining) and combing. Loop interchanging is well known as a method to enhance parallelism and to improve memory hierarchy performance [AlK84,Wol82]; here we discuss a slightly more powerful formulation than the usual pairwise interchanging, namely ***loop circulation*** (interchanging a given loop inwards or outwards over several loops in a single step) [Ban90]. ***Loop skewing*** is a simple transformation that allows a compiler to implement the wavefront method, even in multiple dimensions [Wol86b]. ***Loop reversal*** was first implemented in the compilers for the Texas Instruments' ASC [Wed75]. ***Loop rotation*** has been introduced as a method to map algorithms efficiently onto multiprocessors [Wol90]. ***Sectioning*** is used to divide a loop into fixed size sections for vector processing, or into a fixed number of sections for concurrent execution [All83,Lov77]; *combing* is a variant of sectioning. This paper also discusses how to modify loop limits when interchanging loops that traverse certain simple convex iteration spaces.

2 Overview of Data Dependence

Nested loops define an *iteration space,* comprising a finite discrete Cartesian space with dimensionality equal to the loop nest level. For example, the loop below defines a two-dimensional 5X10 iteration space. In imperative languages, the semantics of a serial loop define the order in which the points in the iteration space are visited.

Program 1:

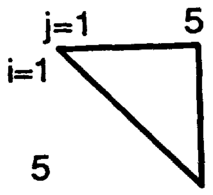
```
    for i = 1 to 5
      for j = 1 to 10
        A(i,j) = B(i,j) + C(i)*D(j)
      endfor
    endfor
```



There is no reason that the iteration space need be rectangular; many popular algorithms have inner loops whose limits depend on the values of outer loop indices. The iteration space for the loop below is triangular, suggesting the name triangular loop. Other interesting iteration space shapes can be defined by nested loops, such as trapezoids, rhomboids, and so on; some of these shapes can be generated from each other via loop restructuring.

Program 2:

```
    for i = 1 to 5
      for j = i to 5
        A(i,j) = B(i,j) + C(i)*D(j)
      endfor
    endfor
```



Data Dependence. Many compilers available for today's advanced computers can detect vector or parallel operations from serial loops. Compilers discover the essential data flow (or data dependence) in the loop and allow vector or parallel execution when the data dependence relations are not violated.

Loop restructuring transformations, such as loop interchanging, are often applied to enhance the available parallelism or otherwise optimize performance; data dependence information is needed to test whether restructuring transformations are legal. Here, we say a transformation is *legal* if normal sequential execution of the restructured program will produce the same answer as the original program; in other words, the sequential loops and the statement ordering must satisfy all dependence relations. When compilers generate code for vector or parallel computers, there may be other means for satisfying dependence relations (such as interprocessor synchronization) which will allow the use of an otherwise 'illegal' transformation.

In imperative languages, there are three essential kinds of data dependence. A *flow-dependence* relation occurs when the value assigned to a variable or array element in the execution of one *instance* of a statement is used by the subsequent execution of an instance of the same or another statement. The loop below has a flow dependence relation from statement S_1 to itself, since the value assigned to `A(i+1)` will be used on the next iteration of the loop, written $S_1 \, \delta \, S_1$.

```
      for i = 1 to N-1
$S_1$:    A(i+1) = A(i) + B(i)
      endfor
```

An *anti-dependence* relation occurs when the value read from a variable or array element in an instance of some statement is subsequently reassigned. In the loop below there is an anti-dependence relation from S_1 to S_2, since B(i,j+1) is used in S_1 and subsequently reassigned by S_2 in the next iteration of the j loop, written $S_1 \bar{\delta} S_2$.

```
      for i = 1 to N
        for j = 1 to M-1
$S_1$:      A(i,j) = B(i,j+1) + 1
$S_2$:      B(i,j) = C(i) - 1
        endfor
      endfor
```

Finally, an *output dependence* relation occurs when some variable or array element is assigned in an instance of a statement and subsequently reassigned. An example of this is shown below where there is an potential output dependence relation from S_2 to S_1, since the variable B(i+1) assigned in S_2 may be reassigned in the next iteration of the loop by S_1, written $S_2\ \delta^o\ S_1$. This also shows that compilers must approximate the data dependence relations in a program; since a compiler can not know the paths that will be taken at run time, it must make conservative assumptions.

```
      for i = 1 to N-1
$S_1$:    if(A(i) > 0) B(i) = C(i)/A(i)
$S_2$:    B(i+1) = C(i) / 2
      endfor
```

Distance and Direction Vectors. In order to apply a wide variety of loop transformations, data dependence relations are annotated with information showing how they are affected by the enclosing loops. Three such annotations are popular today. Many dependence relations have a constant distance in each dimension of the iteration space. When this is the case, a ***distance vector*** can be built where each element is a constant integer representing the dependence distances in the corresponding loop. For example, in the following program there is a data dependence relation in the iteration space as shown; each iteration (i,j) depends on the value computed in iteration (i,j-1). The distances for this dependence relation are zero in the i loop and one in the j loop, written $S_1\ \delta_{(0,1)}\ S_1$.

Program 3:

```
      for i = 1 to N
        for j = 2 to M
$S_1$:      A(i,j) = A(i,j-1) + B(i,j)
        endfor
      endfor
```

Each distance vector will have n entries, where n is the nesting level of the loops surrounding the source and sink of the dependence. Since dependence distances are usually small, short words or perhaps even signed bytes could be used, reducing the storage requirements.

For many transformations, the actual distance in each loop may not be so important as just the sign of the distance in each loop; also, often the distance is not constant in the loop, even though it may always be positive (or always negative). As an example, in the loop:

Program 4:

```
     for i = 1 to N
       for j = 1 to N
S1:      X(i+1,2*j) = X(i,j) + B(i)
       endfor
     endfor
```

the assignment to `X(i+1,2*j)` is used in some subsequent iteration of the `i` and `j` loops by the `X(i,j)` reference. Some of the dependence relations for this program are given in the table below:

	assigned by		used by		dependence
element	i	j	i	j	distance
X(2,2)	1	1	2	2	(1,1)
X(3,4)	2	2	3	4	(1,2)

The distance in the `j` loop for this dependence is always positive, but is not a constant. A common method to represent this is to save a vector of the signs of the dependence distances, called a *direction vector*. Each direction vector element will be one of `{+,0,-}` [Ban88]; for historical reasons, these are usually written `{<,=,>}` [Wol78, WoB87, Wol89]. In Program 4, the direction vector associated with the dependence relation is $S_1\ \delta_{(<,<)}\ S_1$; in Program 3, the dependence relation would be written $S_1\ \delta_{(=,<)}\ S_1$.

Another popular data dependence annotation saves only the nest level of the outermost loop with a non-(=) direction (non-zero distance) [AlK87]. The dependence relation for Program 3 has a zero distance in the outer loop, but a non-zero distance in the inner loop, so this dependence relation is *carried* by the inner `j` loop. Some dependence relations may not be carried by any loop, as below:

```
     for i = 1 to N
       for j = 2 to M
S1:      A(i,j) = B(i,j) + C(i,j)
S2:      D(i,j) = A(i,j) + 1
       endfor
     endfor
```

Here the references to `A(i,j)` produce a dependence relation from S_1 to S_2 with zero distance in both loops. This is written $S_1\ \delta_{(=,=)}\ S_2$ or $S_1\ \delta_{(0,0)}\ S_2$. Since it is carried by neither of the loops, it is called a *loop independent dependence* [AlK87]. This annotation by itself is too coarse for most applications; we will use the notion of loop-carried dependence in our discussion, but we show how to get this information from direction vectors.

3 Direction Vector Extensions and Bit Vectors

Here we extend direction vectors by adding a fourth dependence direction explicitly for reductions. Take, for example, the program:

```
     for i = 1 to N
S1:    S = S + A(i)
     endfor
```

Previous work in data dependence would classify this program as having the dependence relation $S_1\ \delta_{(<)}\ S_1$, thus implicitly preventing reordering of the index set. Because some beneficial transformations change the order of the accumulation, we distinguish associative reductions in the direction vector by using a reduction direction, written: $S_1\ \delta_{(R)}\ S_1$. The difference is more noticeable with nested reductions:

Program 5:
```
        for i = 1 to N
          for j = 1 to M
S1:         S = S + B(i,j)
          endfor
        endfor
```

Previous work would say that this loop has the dependence relations $S_1\ \delta_{(<,*)}\ S_1$ as well as $S_1\ \delta_{(=,<)}\ S_1$. The dependence test for loop interchanging, for example, is that there must be no (<,>) directions; unfortunately, there is a (<,>) implied by the (<,*). With extended direction vectors, there is only the dependence relation $S_1\ \delta_{(R,R)}\ S_1$, reduction in both dimensions. A reduction direction essentially corresponds to a dependence distance of one. Practically speaking, compiler may only be able to find reduction directions when all other directions are (=), and that is the only case dealt with here.

Reduction dependence relations are carried by all loops with an R direction. In Program 5, for instance, the dependence relation is carried by both loops, while in the following program the dependence relation is carried by the i and k loops, written $S_1\ \delta_{(R,=,R)}\ S_1$:

```
        for i = 1 to N
          for j = 1 to M
            for k = 1 to L
S1:         T(j) = T(j) + B(i,j,k)
            endfor
          endfor
        endfor
```

Bit Vectors. In previous work, we discussed the potential of using a bit vector to represent a direction vector, with one bit for each direction in each loop [Wol89]. With the reduction direction, a direction vector would use $4n$ bits to represent a dependence relation between two statements in n nested loops; if n is limited to a small number (like 7 or 15), then each direction vector can be stored in a single 32- or 64-bit word.

As mentioned in [Wol89], taking the union of multiple directions ('ORing' the bit vectors) can save space but with a loss of precision. For instance, ORing the bit vectors for the direction vectors (<,>) and (=,=) gives the combined bit vector $(\leq,\geq)$. The problem is that this is also the sum of

$$(<,>) + (<,=) + (=,=) + (=,>) \rightarrow (\leq,\geq)$$

It thus looks like there is a spurious (<,=) and (=,>) direction; the latter direction can be filtered out by disallowing infeasible directions. However, experience has shown that these spurious directions do not inhibit transformation of the program; since our goal is restructuring through transformations, we accept the imprecision.

Another problem is shown by ORing the bit vectors $(<,=) + (=,<) \rightarrow (\leq,\leq)$; this combined direction could be the sum of

$$(<,<) + (<,=) + (=,<) + (=,=) \rightarrow (\leq,\leq)$$

Here there is a spurious (<,<) and (=,=); the (<,<) will usually cause no problems. However the spurious (=,=) does, since it will prevent reordering of code within the body of the loop, even within a single iteration. Thus, as in [Wol89], we add one additional bit in the direction vector, called the (==) bit, which will be set when the combined direction vector includes the "all-equals" case. For example:

$$(<,<) + (<,=) + (=,<) \rightarrow (\leq,\leq)$$

and

$$(<,<) + (<,=) + (=,<) + (=,=) \rightarrow (\leq,\leq,==)$$

Using bit vectors to represent dependence direction vectors does not preclude the use of efficient techniques to build the direction vector, such as the dependence hierarchy introduced in [BuC86].

We will show how to manipulate the bit vector representation of dependence directions for each program transformation. For instance, the test to see whether the iterations of a loop at nest level k can be executed in parallel is that the loop carries no dependence relations. Using our bit-vector representation, this test can be done by testing for a loop-carried (<) direction, or a (R) direction. Testing for a loop-carried (<) direction is done by building a test direction vector $\Psi_k = (\psi_1, \psi_2, \cdots, \psi_n, \psi_{==})$ where

$$\psi_j = \{=\},\ 1 \leq j < k;\ \psi_k = \{<\};\ \psi_h = \emptyset,\ k < h \leq n;\ \psi_{==} = \emptyset$$

Then, for each data dependence relation in the loop, test whether the corresponding direction vector Φ carries a dependence at nest k with the test:

$$\texttt{carries(k)} := (\Psi_k \bigcap \Phi) = \Psi_k \text{ OR } \psi_k \bigcap \{R\} \neq \emptyset$$

Loop k carries any dependence relation for which `carries` is true. This method has the advantage of building the test vector Ψ only once for the loop and using it many times.

4 Program Restructuring Transformations

The program restructuring transformations we will investigate are

- loop interchanging,
- loop skewing,
- loop reversal,
- loop rotation,
- loop sectioning and combing.

For each transformation, we will describe the conditions under which it is legal (testing the data dependence relations), its effect on data dependence relations, and its effect on the shape of the iteration space. For the discussion below, assume that there are n nested loops, and that every dependence relation is annotated with a direction vector $(\phi_1, \phi_2, \ldots, \phi_n, \phi_{==})$, where each direction vector element $\phi_j \subseteq \{<,=,>,R\}$ and is represented by four bits, except that $\phi_{==} \subseteq \{==\}$ and is represented by a single additional bit.

Loop Interchanging. Interchanging nested loops can dramatically change the execution characteristics of a loop, and can enhance the parallelism available at the inner or outer loop levels [AlK84, Wol78, Wol82, Wol89]. In a loop of the following form:

```
for i_1 = ...
 for i_2 = ...
   . . .
    for i_k = ...
     for i_{k+1} = ...
       . . .
        for i_n = ...
          computation
        endfor
     . . .
endfor
```

the i_k and i_{k+1} loops can legally be interchanged only when there is no data dependence relation in the computation that is *carried* by the i_k loop which also has a $(>)$ direction in the i_{k+1} loop; this corresponds to a direction vector of $(=,=,\ldots,=,=,<,>,*,*,\ldots)$.

Interchanging adjacent loops essentially transposes an iteration space about the major diagonal; thus a 5×10 iteration space (Program 1) becomes a 10×5 space after interchanging, and an upper right triangular iteration space (Program 2) becomes a lower left triangle.

A more general form of loop interchanging, called a *circulation* in [Ban90], can interchange loop i_k inwards to inside any loop (up to i_n) or outwards to outside any enclosing loop (down to i_1) in a single step; we will call these *in-circulation* and *out-circulation.* A sufficient (but not necessary) condition to test whether loop i_k can be in-circulated inside of loop i_m (assuming the loops are all tightly nested and $m > k$), is that for every dependence relation between statements in the loop, any one of the following three conditions hold:

(a) There is no $(<)$ component in the direction vector element for i_k; this is equivalent to $\{<\}\bigcap\phi_k=\emptyset$.

(b) The dependence relation is carried by one of the outer loops; this is equivalent to $\exists j\in\{1\cdots k-1\}$ s.t. $\{=\}\bigcap\phi_j=\emptyset$.

(c) There is no $(>)$ component in the direction vector element for loops over which i_k is being interchanged; this is equivalent to $\{>\}\bigcap\phi_h=\emptyset,\ \forall h\in\{k+1\cdots m\}$.

Using a bit vector for the direction vector, conditions (a) and (b) can be tested with a single bit vector test; first, construct a direction bit vector: $\Psi=(\psi_1,\cdots,\psi_n)$ where

$$\psi_j=\{=\},\ 1\leq j<k;\ \psi_k=\{<\};\ \psi_h=\emptyset,\ k<h;\ \psi_{==}=\emptyset$$

Then for all direction vectors Φ, test $\Psi\bigcap\Phi=\Psi$. If the equality does not hold, then either the dependence relation is carried by an outer loop (test b) or is not carried by the k loop (test a), and interchanging is legal; otherwise condition (c) must be tested.

Condition (c) can be tested with another bit vector test; construct a second test vector Ψ where

$$\psi_h=\{>\},\ k<h\leq m;\ \psi_j=\emptyset,\ \text{otherwise}$$

Then for any direction vector Φ, test $\Psi\bigcap\Phi=\emptyset$. If the equality holds, then there are no $(>)$ directions to worry about and in-circulation is legal.

This test is not precise, in the sense that there can be dependence relations that fail both tests but still allow the circulation. For instance, in 3 nested loops, we might have a dependence relation with a direction vector $(<,<,>)$; to in-circulate the outermost loop i_1 inside of the innermost loop i_3, we find that the first test fails (since the dependence is in fact carried by i_1), and the second test also fails (since there is in fact a $(>)$ component in ϕ_3. Yet, in-circulation is legal in this case [Ban90]. A precise dependence test for in-circulation requires modifying condition (c) above with m-k conditions, of the form:

(c') For each p such that $k<p\leq m$, either there is no (>) component in the ϕ_p, or there is no (=) component in some ϕ_q for $k<q<p$; this is equivalent to $\{=\}\bigcap\phi_q=\emptyset,\ \forall q\in\{k+1\cdots p-1\}$ or $\{>\}\bigcap\phi_p=\emptyset$.

This condition can be tested for some value of p by constructing the test vector Ψ_p where

$$\psi_q=\{=\},\ k<q<p;\ \psi_p=\{>\};\ \psi_j=\text{empty},\ \text{otherwise}$$

Then for any direction vector Φ, test $\Psi_p\bigcap\Phi=\Psi_p$ for all p such that $k<p\leq m$. If the equality does not hold, then either ϕ_p does not have a (>) component, or if it does, some loop between i_k and i_p has a (<) direction that will carry the dependence by the time i_k is interchanged up to the i_p loop; in some sense, a direction of (<) "protects" any (>) directions to the right in the direction vector. If the equality holds, then there is an unprotected (>) direction to worry about and in-circulation is illegal. Since this test requires m-k bit vector operations, the simpler condition (c) should be tested first to screen the trivial cases.

A necessary and sufficient condition to test whether loop i_k can be out-circulated to outside of loop i_m ($m < k$, again assuming the loops are tightly nested), is that for every dependence relation between statements in the loop, any one of the following three conditions hold:

(a) There is no (>) component in the direction vector element for i_k; this is equivalent to $\{>\}\bigcap\phi_k=\emptyset$.

(b) The dependence relation is carried by one loops surrounding m; this is equivalent to $\exists j\in\{1\cdots m-1\}$ s.t. $\{=\}\bigcap\phi_j=\emptyset$.

(c) There is no (<) component in the direction vector element for loops over which i_k is being interchanged; this is equivalent to $\{<\}\bigcap\phi_h=\emptyset,\ \forall h\in\{m\cdots k-1\}$.

Again, conditions (a) and (b) can be tested with a single bit vector test by constructing a direction bit vector Ψ where

$$\psi_j=\{=\},\ 1\leq j<m;\ \psi_k=\{>\};\ \psi_h=\emptyset,\ m\leq h,\ h\neq k;\ \psi_{==}=\emptyset$$

Then for any direction vector Φ, test $\Psi\bigcap\Phi=\Psi$. If the equality does not hold, then either the dependence relation is carried by an outer loop (test b) or there is no (>) in the k loop (test a), and interchanging is legal.

Condition (c) can be tested with another bit vector test; construct the direction bit vector Ψ where

$$\psi_h=\{<\},\ m\leq h<k;\ \psi_j=\emptyset,\ \text{otherwise}$$

Then for any direction vector Φ, test $\Psi\bigcap\Phi=\emptyset$. If the equality holds, then there are no (<) directions to worry about and out-circulation is legal. Note that testing for out-circulation can be done precisely with just two bit vector operations.

In-circulation and out-circulation affect the direction vectors in the obvious manner (moving ϕ_k to the m-th position, and shifting ϕ_{k+1} through ϕ_m or ϕ_m through ϕ_{k-1} appropriately); this can be done on the bit vector representation with a few mask and shift operations.

Notice that since we are allowing the order of reductions to change, reduction dependences do not prevent interchanging. The reduction loops in Program 5, for instance, could be interchanged in our scheme.

Loop Skewing. Loop skewing was introduced as a alternate derivation of the wavefront method [Wol86b]. In the two-nested case below:

```
for i = 2 to N-1
 for j = 2 to N-1
  computation(i,j)
 endfor
endfor
```

the j loop can be skewed with respect to the i loop by adding i to the upper and lower limits of the j loop and subtracting i from j within the body of the loop:

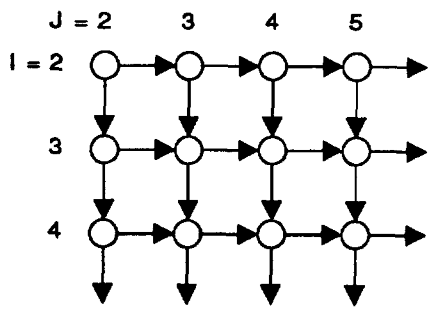
```
for i = 2 to N-1
 for j = i+2 to i+N-1
  computation(i,j-i)
 endfor
endfor
```

In general, a loop can be skewed with respect to any outer loop in which it is contained, and the loop can be skewed with any integer factor. For this paper, the skewing factor is restricted to be +1 or -1 (the example above uses a factor of +1; a factor of -1 would subtract i from the loop limits and add i within the body of the loop).

Loop skewing itself is always legal. By itself it has no effect on the order of execution of the iterations of the loop. However it has significant impact on the data dependence relations in the loop. For instance, the following loop:

```
for i = 2 to N-1
 for j = 2 to N-1
  A(i,j) = A(i-1,j) + A(i,j-1)
 endfor
endfor
```

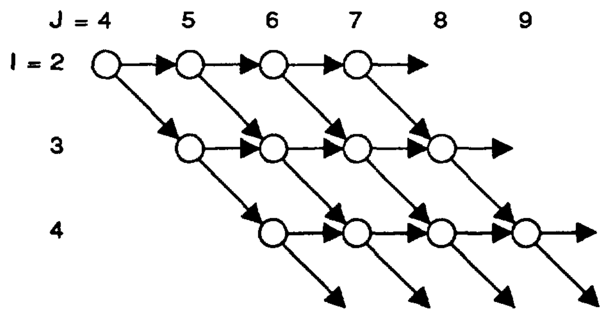
has two flow-dependence relations, with direction vectors (<,=) and (=,<). The dependence relations in the iteration space are:



Since each loop carries a dependence relation, neither loop can be executed in parallel. However, skewing the inner loop by a factor of +1 changes the program to:

```
for i = 2 to N-1
 for j = i+2 to i+N-1
  A(i,j-i) = A(i-1,j-i) + A(i,j-i-1)
 endfor
endfor
```

The new iteration space is a parallelogram:



In particular, the dependence relations have direction vectors (<,<) and (=,<). Now consider what happens when the skewed loops are interchanged [Wol86a]:

```
for j = 2+2 to N-1+N-1
 for i = max(2,j-N+1) to min(N-1,j-2)
  A(i,j-i) = A(i-1,j-i) + A(i,j-i-1)
 endfor
endfor
```

Now the dependence direction vectors are (<,<) and (<,=), meaning that the outer j loop carries both dependence relations. This means that the inner loop can be executed in parallel, since it no longer carries any dependence relations. This is the main effect of which compilers will take advantage by loop skewing.

As already mentioned, loop skewing is always legal. As with loop interchanging, loop skewing can change the direction vector associated with each dependence relation in the loop. This paper uses only factors of ±1; to find cases where other factors are useful, dependence distance vectors are necessary. Skewing the m loop with respect to the k loop by a positive or negative factor will change direction vector element ϕ_m according to Table 1. Since a direction vector has sign but no magnitude information, this table corresponds to the sign of the result of adding two numbers of indefinite magnitude (positive skew) or subtracting two numbers of indefinite magnitude (negative skew). Adding two positive numbers is always positive, but adding a positive and negative number results in a number with indefinite sign; this gives the "*" entries in the table. The "*" corresponds to the set {<,=,>}.

positive skew		ϕ_m				negative skew		ϕ_m			
		<	=	>	R			<	=	>	R
	<	<	<	*			<	*	>	>	
ϕ_k	=	<	=	>	R	ϕ_k	=	<	=	>	R
	>	*	>	>			>	<	<	*	
	R		<		R		R		>		R

Table 1. New value of ϕ_m after loop skewing.

Reduction directions require special handling; remember that we allow reduction directions only when all other directions are (=). A direction vector with both ϕ_k=(R) and ϕ_m=(R) will remain unchanged by skewing. However, if $(\text{phi}_k,\phi_m)=(R,=)$, then we have to treat this like the case of $(\text{phi}_k,\phi_m)=(<,=)$, and change ϕ_k to (<) also. If there are other reduction directions in the direction vector, then those must also be appropriately modified.

Loop skewing can be used to find additional parallelism when it can change a zero or negative direction in the inner loop to a positive direction, thus allowing loop interchanging to make that the dependence-carrying loop.

Loop Reversal. Loop reversal is simply running a loop backwards; it consists of switching the lower and upper limits and negating the increment of the loop. It has little applicability when compiling imperative languages, except for the rare cases when it can be used to change (>) directions to (<) directions to allow loop interchanging. It is useful when translating applicative programs to imperative semantics. In particular, when an applicative program requires a loop to run backwards, as in the program below:

```
for i in 1:N-1
 A(i) = A(i+1) + 1
endfor
```

reversal is required to allow normal imperative semantics to execute the loop sequentially.

Reversing a loop is legal if it does not carry any dependence relations other than reduction relations. The legality of reversal of the k loop is tested by finding whether there are any dependence carrying (<) directions at nest k. The compiler can build a test vector Ψ where

$$\psi_j=\{=\},\ 1\leq j<k;\ \psi_k=\{<\};\ \psi_h=\emptyset,\ k<h;\ \psi_{==}=\emptyset$$

Then for all direction vectors Φ, test $\Psi\bigcap\Phi=\Psi$. If the test is true, then loop k carries a data dependence relation that would be violated by reversal.

Reversing the k loop will change the direction vector as:

$$(\phi_1,\phi_2,\ldots,\phi_{k-1},\phi_k,\phi_{k+1},\ldots,\phi_n) \xrightarrow{reverse} (\phi_1,\phi_2,\ldots,\phi_{k-1},-\phi_k,\phi_{k+1},\ldots,\phi_n)$$

where $-\phi_k$ is defined by Table 2. This modification can be managed on the bit vector by some simple masking operations.

	ϕ_k			
	<	=	>	R
$-\phi_k$	>	=	<	R

Table 2. New value of ϕ_k after loop reversal.

Loop Rotation. Loop rotation is most easily defined when the loops are "normalized", that is have a lower limit of zero and increment of one; unnormalized loops can always be normalized by a simple program transformation if desired [AlK87,KKP81]. Rotation is much like skewing [Wol90]; in the two-nested case below:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i,j)
 endfor
endfor
```

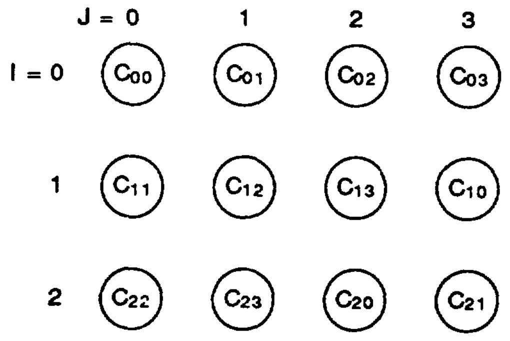
rotating the j loop with respect to the i loop by a factor of +1 changes the computation:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i, (j-i) mod M)
 endfor
endfor
```

while rotation by a factor of -1 produces:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i, (j+i) mod M)
 endfor
endfor
```

Loop rotation corresponds to skewing the loop around a torus; the picture below shows a rotated iteration space (by factor of -1) where C_{ij} corresponds to *computation*(i, j), and N=3, M=4.



Loop rotation can be applied to non-adjacent loops, as with loop skewing. Unlike skewing, rotation has no effect on the shape of the iteration space, since it merely shifts the iteration space around.

Also unlike skewing, rotation is not always legal. Except for special cases dealing with reduction directions (as in skewing), rotating the m loop with respect to the outer k loop will leave ϕ_k unchanged, since the order of execution of the iterations of the k loop are unchanged. An entry ϕ_m of (<) or (>) will become (*) however, since any dependence relation can potentially "wrap around" the iteration space and point in any direction. Thus any (<) dependence carried by the m loop will be violated by loop rotation, the same condition as for loop reversal. As with loop skewing, handling a direction vector $(\phi_k, \phi_m) = (R, =)$ (reduction direction in the outer loop, non-reduction in rotated loop), requires treating this like the case $(\phi_k, \phi_m) = (<, =)$; this is because a reduction direction is only well defined when all other directions are (=). Except for the cases in Table 3, rotation of i_m with respect to i_k will change direction vector elements (ϕ_k, ϕ_m) to $(\phi_k, *)$.

(ϕ_k,ϕ_m) positive rotation		ϕ_m		(ϕ_k,ϕ_m) negative rotation		ϕ_m	
		=	R			=	R
ϕ_k	=	(=,=)	(=,R)	ϕ_k	=	(=,=)	(=,R)
	R	(<,*)	(R,R)		R	(<,*)	(R,R)

Table 3. New value of ϕ_m after rotation; other cases have "*" value.

It is also well defined to rotate a loop with respect to an inner loop. The simple two-nested computation:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i,j)
 endfor
endfor
```

would be changed to:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation((i±j) mod M, j)
 endfor
endfor
```

As before, after rotating outer loop m with respect to inner loop k, ϕ_k will remain unchanged and ϕ_m will become (*), except the cases shown in Table 4. Again, any non-reduction dependence relation carried by the m loop will be violated by loop rotation.

(ϕ_m,ϕ_k) positive rotation		ϕ_k		(ϕ_m,ϕ_k) negative rotation		ϕ_k	
		=	R			=	R
ϕ_m	=	(=,=)	(*,<)	ϕ_m	=	(=,=)	(*,<)
	R	(R,=)	(R,R)		R	(R,=)	(R,R)

Table 4. New value of ϕ_m after rotation with respect to inner loop.

Sectioning. Sectioning (strip mining) is used by vectorizing compilers to fix the size of the innermost vector loop to the maximum hardware vector length [All83,Lov77]. It is also used to divide a loop into a fixed number of equal size chunks to spread the work across multiple processors. Sectioning corresponds to simply splitting the a loop into two adjacent nested loops, where the inner loop (the element loop) iterates along the elements of a single section, and the outer loop (the section loop) moves to the next section. Thus the loop:

```
for i = 1 to N
  computation(i)
endfor
```

could be sectioned (to a maximum size of 32) by changing the loop to:

```
ss = 32
for is = 0 to N-1 by ss
  for i = is+1 to MIN(N,is+ss)
    computation(i)
  endfor
endfor
```

or it could be sectioned into 8 equal size sections by:

```
ss = (N+7)/8
for is = 0 to N-1 by ss
  for i = is+1 to MIN(N,is+ss)
    computation(i)
  endfor
endfor
```

Sectioning is always legal, since by itself it has no effect on the order of the execution of the iterations of the loop. It does change the number of loops, thus adding a dimension to the iteration space (though the total volume remains the same) and adding an element to the direction vectors. Sectioning the k loop will transform each dependence relation into one or two dependence relations. Sectioning loop k will change the direction vector as:

$$(\phi_1, \phi_2, \ldots, \phi_{k-1}, \phi_k, \phi_{k+1}, \ldots, \phi_n) \xrightarrow{\text{sect}(k)} (\phi_1, \phi_2, \ldots, \phi_{k-1}, \phi_k^s, \phi_k^e, \phi_{k+1}, \ldots, \phi_n)$$

where (ϕ_k^s, ϕ_k^e) are defined Table 5.

	ϕ_k			
	<	=	>	R
(ϕ_k^s, ϕ_k^e)	(=,<) (<,*)	(=,=)	(=,>) (>,*)	(R,R)

Table 5. New value of direction vector after sectioning.
To see why sectioning a loop with (<) or (>) dependences produces into two different directions, let us examine the loop:

```
     for I = 2 to N
S1:    A(I) = A(I-3) + A(I-4) + A(I-5) + B(I)
     endfor
```

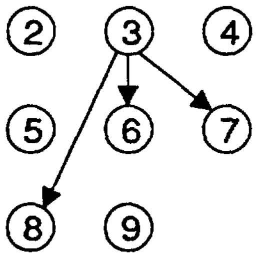
There are three flow dependence relations with dependence distances 3, 4 and 5; however, they would all have the same direction vector $S_1 \; \delta_{(<)} \; S_1$. The dependence relations from iteration I=3 would be:

$$\begin{tikzcd}
2 & 3 \arrow[rrr, bend left] \arrow[rrrr, bend left] \arrow[rrrrr, bend left] & 4 & 5 & 6 & 7 & 8 & 9
\end{tikzcd}$$

Sectioning this loop to a section size of 3 produces the program:

```
      for Is = 4 to N-1 by 3
        for I = Is+1 to MIN(N,Is+3)
S1:     A(I) = A(I-3) + A(I-4) + A(I-5) + B(I)
        endfor
      endfor
```

The dependence relations in the iteration space for I=3 are now:



If the dependence distances are known, then the exact distance or direction after sectioning can be computed; since our scheme only uses direction vectors, it must assume the worst case.

Combing. Combing is a modification of sectioning where essentially the section and element loops are interchanged. After sectioning with a section size of ss, each section loop comprises ss consecutive iterations of the loop. On the other hand, after combing to get cc combs, each comb comprises every cc^{th} iteration. Combing is useful for the same reasons as sectioning, but especially when assigning iterations to multiple processors when load balancing or inter-processor dependence relations must be satisfied. Thus the loop:

```
for i = 1 to N
  computation(i)
endfor
```

can be combed into 8 equal size combs by:

```
cc = 8
for ic = 1 to cc
  for i = ic to N by cc
    computation(i)
  endfor
endfor
```

Whereas sectioning is always legal, combing is not. Like sectioning, combing increases the number of loops and adds a dimension to the iteration space and an element to the direction vectors. Combing the k loop will transform each dependence relation into one or two dependence relations. Combing loop k will change the direction vector as:

$$(\phi_1, \phi_2, \ldots, \phi_{k-1}, \phi_k, \phi_{k+1}, \ldots, \phi_n) \xrightarrow{comb(k)} (\phi_1, \phi_2, \ldots, \phi_{k-1}, \phi_k^{\bullet}, \phi_k^{\bullet}, \phi_{k+1}, \ldots, \phi_n)$$

where $(\phi_k^{\bullet}, \phi_k^{\bullet})$ are defined by Table 6. Since combing corresponds to interchanging the section and element loops, it is not surprising that these are simply the same direction vectors for sectioning, with the entries interchanged. Like rotation and reversal, any non-reduction dependence relation carried by the k loop will be violated by combing.

	ϕ_k			
	<	=	>	R
$(\phi_k^{\bullet}, \phi_k^{\blacksquare})$	(<,=) (*,<)	(=,=)	(>,=) (*,>)	(R,R)

Table 6. New value of direction vector after combing.

5 Loop Interchanging Convex Iteration Spaces

Loop interchanging of triangular and trapezoidal loops is discussed in previous work [Ban90, Wol86a, Wol89]. This section presents a set of general rules for interchanging loops that traverse simple convex iteration spaces. The convex iteration spaces handled by this methods are described with the loop limits:

```
for $I_1$ = max( $L_{1,1,0}$, $L_{1,2,0}$, ... ) to min( $U_{1,1,0}$, $U_{1,2,0}$, ... )
 for $I_2$ = max( $L_{2,1,0}+L_{2,1,1}I_1$, ... ) to min( $U_{2,1,0}+U_{2,1,1}I_1$, ... )
  . . .
   for $I_k$ = max( $L_{k,1,0}+\sum_{j=1}^{k-1}L_{k,1,j}I_j$, ... ) to min( $U_{k,1,0}+\sum_{j=1}^{k-1}U_{k,1,j}I_j$, ... )
```

where each **max** and **min** has a list of one or more arguments of the form given, and a **max** or **min** with a single argument returns the value of that single argument. The parameters $L_{k,1,j}$ and $U_{k,1,j}$ are integer constants. The method described here modifies the loop limits when adjacent loops are interchanged; loop circulation or other multiple loop interchanges can proceed by multiple applications of this method. These forms of loop limits sometimes arise when multiple transformations (such as skewing and interchanging) are applied to nested loops; it is useful to have a general framework to deal with these limits.

Assume we are interchanging two adjacent loops in the above program; each loop limit is an extremum of an affine expression of outer loop limits. From the perspective of the inner loop, the outer loop limit expressions are invariants, while from the perspective of the outer loop, the inner loop limits are an invariant expression plus a constant times the outer loop index:

```
for I = max( $L_{I,1}$, $L_{I,2}$, ... ) to min( $U_{I,1}$, $U_{I,2}$, ... )
 for J = max( $L_{J,1}+L_{J,1,I}I$, $L_{J,2}+L_{J,2,I}I$, ... )
                 to min( $U_{J,1}+U_{J,1,I}I$, $U_{J,2}+U_{J,2,I}I$, ... )
```

We present this as a pseudo-algorithm. $\mathbf{L_I}$ will be a set of expressions, the maximum of which will represent the lower limit of the I loop (inner loop after interchanging), while $\mathbf{U_I}$ will be a set of expressions, the minimum of which will be the upper limit; similarly for $\mathbf{L_J}$ and $\mathbf{U_J}$ (outer loop after interchanging). The algorithm manages each of these as a set.

1. Initialize by setting

$$\mathbf{L_I} \leftarrow \{L_{I,1}, L_{I,2}, \cdots\}$$

$$\mathbf{U_I} \leftarrow \{U_{I,1}, U_{I,2}, \cdots\}$$

$$\mathbf{L_J} \leftarrow \emptyset$$

$$\mathbf{U_J} \leftarrow \emptyset$$

2. For each lower limit expression $L_{J,n}+L_{J,n,I}I$ in the original J loop, do one of steps 2a, 2b or 2c depending on the sign of $L_{J,n,I}$.

   2a. If $L_{J,n,I} = 0$, set:

$\mathbf{L_J \leftarrow L_J} \bigcup \{L_{J,n}\}$

2b. If $L_{J,n,I} > 0$, set:

$\mathbf{L_J \leftarrow L_J} \bigcup \{L_{J,n}+L_{J,n,I}L_{I,k} \mid \forall k\}$

$\mathbf{U_I \leftarrow U_I} \bigcup \{\left\lfloor \dfrac{J-L_{J,n}}{L_{J,n,I}} \right\rfloor\}$

2c. If $L_{J,n,I} < 0$, set:

$\mathbf{L_J \leftarrow L_J} \bigcup \{L_{J,n}+L_{J,n,I}U_{I,k} \mid \forall k\}$

$\mathbf{L_I \leftarrow L_I} \bigcup \{\left\lceil \dfrac{L_{J,n}-J}{-L_{J,n,I}} \right\rceil\}$

3. For each upper limit expression $U_{J,n}+U_{J,n,I}I$ in the original J loop, do one of steps 3a, 3b or 3c depending on the sign of $U_{J,n,I}$.

3a. If $U_{J,n,I} = 0$, set:

$\mathbf{U_J \leftarrow U_J} \bigcup \{U_{J,n}\}$

3b. If $U_{J,n,I} > 0$, set:

$\mathbf{U_J \leftarrow U_J} \bigcup \{U_{J,n}+U_{J,n,I}U_{I,k} \mid \forall k\}$

$\mathbf{L_I \leftarrow L_I} \bigcup \{\left\lceil \dfrac{J-U_{J,n}}{U_{J,n,I}} \right\rceil\}$

3c. If $U_{J,n,I} < 0$, set:

$\mathbf{U_J \leftarrow U_J} \bigcup \{U_{J,n}+U_{J,n,I}L_{I,k} \mid \forall k\}$

$\mathbf{U_I \leftarrow U_I} \bigcup \{\left\lfloor \dfrac{U_{J,n}-J}{-U_{J,n,I}} \right\rfloor\}$

This subsumes all previous work on interchanging triangular or trapezoidal loops. A simple example follows:

```
for I = 0 to 4
 for J = 0 to min(I+1, 7-I)
```

The iteration space is:

```
   J = 0 1 2 3 4 5
I= 0   o o
   1   o o o
   2   o o o o
   3   o o o o o
   4   o o o o
```

The interchanged loops have (by the rules above) the limits:

```
for J = 0 to min(5, 7)
 for I = max(0, J-1) to min(4, 7-J)
```

The inner loop limits are:

```
J = 0, I = max(0,-1) to min(4, 7), I = 0 to 4
J = 1, I = max(0, 0) to min(4, 6), I = 0 to 4
J = 2, I = max(0, 1) to min(4, 5), I = 1 to 4
J = 3, I = max(0, 2) to min(4, 4), I = 2 to 4
J = 4, I = max(0, 3) to min(4, 3), I = 3 to 3
J = 5, I = max(0, 4) to min(4, 2), I = 4 to 2
```

Note that for J = 5, the inner loop executes no iterations. An optimization might notice this and optimize the outer loop limits down to J = 0, 4.

Another example; let's try to describe the iteration space:

```
   J = 0 1 2 3 4 5 6 7 8 9
I= 1           o o
   2       o o o o o o o
   3   o o o o o o o o o o
   4   o o o o o o o o o o
   5   o o o o o o o o
   6     o o o o o
   7         o
```

The original loop limits are:

```
for I = 1 to 7
  for J = max(6-2*I, 0, 2*I-11) to min(3*I+2, 9, 17-2*I)
```

Interchanging according to our rules, in steps:

1. Initialize:

$$L_I \leftarrow \{1\}$$

$$U_I \leftarrow \{7\}$$

$$L_J \leftarrow \varnothing$$

$$U_J \leftarrow \varnothing$$

2.1 The first lower limit 6-2*I has a negative factor (-2), so use rule 2c:

$$L_J \leftarrow \{6-2*7\}$$

$$L_I \leftarrow \{1, \left\lceil \frac{6-J}{2} \right\rceil\}$$

2.2 The second lower limit 0 is a constant, so use rule 2a:

$$L_J \leftarrow \{-8, 0\}$$

2.3 The third lower limit 2*I-11 has a positive factor, so use rule 2b:

$$L_J \leftarrow \{-8, 0, -11+2*1\}$$

$$U_I \leftarrow \{7, \left\lfloor \frac{J+11}{2} \right\rfloor\}$$

3.1 The first upper limit 3*I+2 has a positive factor, so use rule 3b:

$$U_J = \{2+3*7\}$$

$$L_I \leftarrow \{1, \left\lceil \frac{6-J}{2} \right\rceil, \left\lceil \frac{J-2}{3} \right\rceil\}$$

3.2 The second upper limit 9 is constant, so use rule 3a:

$\mathbf{U_J}$={23,9}

3.3 The third upper limit 17-2*I has a negative factor, so use rule 3c:

$\mathbf{U_J}$={23,9,17-2*1}

$\mathbf{U_I} \leftarrow \{7, \left\lfloor \frac{J+11}{2} \right\rfloor, \left\lfloor \frac{17-J}{2} \right\rfloor\}$

The final sets are:

$\mathbf{L_J} = \{-8, 0, -8\}$

$\mathbf{U_J} = \{23, 9, 15\}$

$\mathbf{L_I} = \{1, \left\lceil \frac{6-J}{2} \right\rceil, \left\lceil \frac{J-2}{3} \right\rceil\}$

$\mathbf{U_I} = \{7, \left\lfloor \frac{J+11}{2} \right\rfloor, \left\lfloor \frac{17-J}{2} \right\rfloor\}$

corresponding to the loops:

```
for J = max( -8, 0, -8 ) to min( 23, 9, 15 )
  for I = max( 1, $\lceil \frac{6-J}{2} \rceil$, $\lceil \frac{J-2}{3} \rceil$ ) to min( 7, $\lfloor \frac{J+11}{2} \rfloor$, $\lfloor \frac{17-J}{2} \rfloor$ )
```

or, simplifying the J limits:

```
for J = 0 to 9
  for I = max( 1, $\lceil \frac{6-J}{2} \rceil$, $\lceil \frac{J-2}{3} \rceil$ ) to min( 7, $\lfloor \frac{J+11}{2} \rfloor$, $\lfloor \frac{17-J}{2} \rfloor$ )
```

Note that indeed the inner loop limits are correct:

```
J=0, I = max(1,⌈6-0/2⌉,⌈0-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 3 to 5
J=1, I = max(1,⌈6-1/2⌉,⌈1-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 3 to 6
J=2, I = max(1,⌈6-2/2⌉,⌈2-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 2 to 6
J=3, I = max(1,⌈6-3/2⌉,⌈3-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 2 to 7
J=4, I = max(1,⌈6-4/2⌉,⌈4-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 1 to 6
J=5, I = max(1,⌈6-5/2⌉,⌈5-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 1 to 6
J=6, I = max(1,⌈6-6/2⌉,⌈6-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 2 to 5
J=7, I = max(1,⌈6-7/2⌉,⌈7-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 2 to 5
J=8, I = max(1,⌈6-8/2⌉,⌈8-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 2 to 4
J=9, I = max(1,⌈6-9/2⌉,⌈9-2/3⌉ ) to min(7,⌊J+11/2⌋,⌊17-J/2⌋), I = 3 to 4
```

As a final example, let us try to interchange the IJK loop below to a KJI form:

```
for I = 1 to 5
  for J = 6-I to 10
    for K = 1 to 10-I
```

We proceed by first going to the JIK form (using rules 2c and 3a):

```
for J = 6-5 to 10
  for I = max(1,6-J) to 5
    for K = 1 to 10-I
```

then to the JKI form (rules 2a and 3c):

```
for J = 1 to 10
  for K = 1 to min(10-1,10-(6-J))
    for I = max(1,6-J) to min(5,10-K)
```

and finally to the desired KJI form (rules 2a, 3a and 3b):

```
for K = 1 to min(10-1,4+10)
  for J = max(1,K-4) to 10
    for I = max(1,6-J) to min(5,10-K)
```

Another way would be to go from the IJK form first to the IKJ form (rules 2a and 3a):

```
for I = 1 to 5
  for K = 1 to 10-I
    for J = 6-I to 10
```

then to the KIJ form (rules 2a and 3c):

```
for K = 1 to 10-1
  for I = 1 to min(5,10-K)
    for J = 6-I to 10
```

and finally to the KJI form (rules 2c and 3a):

```
for K = 1 to 10-1
  for J = max(6-5,6-(10-K)) to 10
    for I = max(1,6-J) to min(5,10-K)
```

We leave it to the reader to show that all these forms traverse the same iteration space.

6 Summary

Program restructuring is an important capability for compilers and programming tools for complex computer architectures. It is one method to achieve improved performance on advanced computer architectures. Restructuring can never duplicate or surpass the benefits of finding an appropriate algorithm, but the ability to efficiently test for and perform many restructuring transformations will allow compilers to do a better job of mapping programs onto machines.

This paper discusses annotating data dependence relations with a bit vector to represent the direction vector. A new direction vector element was introduced for reductions. The use of the bit vector and especially the reduction direction was shown by describing data dependence tests for several program transformations. This included tests for a particular form of loop interchanging, loop circulation, using only a few bit vector operations per dependence relation. We also discuss how to interchange loops with simple convex loop limits. Some transformations are useful to uncover additional parallelism, while others are useful to optimize performance by taking advantage of certain architectural features. We believe this work will be useful when implementing compilers and other programming tools for advanced parallel computers.

References

[Al183] J. R. Allen, Dependence Analysis for Subscripted Variables and Its Application to Program Transformations, Ph.D. Thesis, Rice University, Houston, TX, April, 1983.

[AlK84] J. R. Allen and K. Kennedy, Automatic Loop Interchange, in *Proc. of the SIGPLAN 84 Symposium on Compiler Construction,* New York, June 1984, 233-246.

[AlK87] J. R. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, *ACM Transactions on Programming Languages and Systems 9*, 4 (October 1987), 491-542.

[Ban88] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.

[Ban90] U. Banerjee, A Theory of Loop Permutations, in *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau and D. Padua (ed.), Pitman, London, 1990, 54-74.

[BuC86] M. Burke and R. Cytron, Interprocedural Dependence Analysis and Parallelization, in *Proc. of the SIGPLAN Notices 86 Symp. on Compiler Construction*, Palo Alto, CA, June 25-27, 1986, 162-175.

[KKP81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe, Dependence Graphs and Compiler Optimizations, in *Conf. Record of the 8th ACM Symp. on the Principles of Programming Languages*, Williamsburg, VA, 1981, 207-218.

[Lov77] D. Loveman, Program Improvement by Source-to-Source Transformation, *J. of the ACM 20*, 1 (January 1977), 121-145.

[Wed75] D. Wedel, Fortran for the Texas Instruments ASC System, *SIGPLAN Notices 10*, 3 (March 1975), 119-132.

[Wol78] M. Wolfe, Techniques for Improving the Inherent Parallelism in Programs, UIUCDCS-R-78-929, Univ. of Illinois, July 1978.

[Wol82] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Ph.D. Thesis, Univ. of Illinois UIUCDCS-82-1105, Urbana, IL, October 1982. (UMI 83-03027).

[Wol86a] M. Wolfe, Advanced Loop Interchanging, in *Proc. of the 1986 Intl Conf. on Parallel Processing*, K. Hwang, S. M. Jacobs and E. E. Swartzlander (ed.), St. Charles, IL, August 19-22, 1986, 536-543.

[Wol86b] M. Wolfe, Loop Skewing: The Wavefront Method Revisited, *Intl J. Parallel Programming 15*, 4 (August 1986), 279-294.

[WoB87] M. Wolfe and U. Banerjee, Data Dependence and Its Application to Parallel Processing, *Intl Journal of Parallel Programming 16*, 2 (April 1987), 137-178.

[Wol89] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman Publishing, London, 1989.

[Wol90] M. Wolfe, Loop Rotation, in *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau and D. Padua (ed.), Pitman, London, 1990, 531-553.

This work was supported by NSF Grant CCR-8906909 and DARPA Grant MDA972-88-J-1004.