International Journal of Parallel Programming, Vol. 15, No. 4, 1986

Loop Skewing: The Wavefront Method Revisited

Michael Wolfe[1,2]

Received November 1986; Accepted February 1987

Loop skewing is a new procedure to derive the wavefront method of execution of nested loops. The wavefront method is used to execute nested loops on parallel and vector computers when none of the loops can be done in vector mode. Loop skewing is a simple transformation of loop bounds and is combined with loop interchanging to generate the wavefront. This derivation is particularly suitable for implementation in compilers that already perform automatic detection of parallelism and generation of vector and parallel code, such as are available today. Loop normalization, a loop transformation used by several vectorizing translators, is related to loop skewing, and we show how loop normalization, applied blindly, can adversely affect the parallelism detected by these translators.

KEY WORDS: Parallelism detection; wavefront method; vectorization; loop transformations; compiler optimization.

1. INTRODUCTION

This paper presents a new derivation of the wavefront method that is particularly suitable for automatic generation by compilers. A new transformation, called *loop skewing*, is introduced and used in combination with loop interchanging in a fashion that will generate code that executes nested loops in the wavefront method. Loop skewing is a simple modification of the shape of the do loop iteration space. Since loop interchanging has already been implemented by several commercial compilers, the addition of simple loop skewing will allow these compilers to implement wavefronting automatically.

[1] Kuck and Associates, Inc., 1808 Woodfield Drive, Savoy, Illinois 61874.
[2] Dept. of Computer Science, University of Illinois, Urbana, Illinois 61801.

0885-7458/86/0800-0279$05.00/0 © 1986 Plenum Publishing Corporation

828/15/4-1

The next section of this paper reviews many of the terms that are used throughout the paper, such as data dependence and loop interchanging. Short examples of interchanging of triangular and trapezoidal loops are shown, since they relate directly to the following section. Next we present the main result; here we introduce loop skewing and show how it is used with loop interchanging to generate code for the wavefront method. Following that we discuss loop normalization, another simple transformation that is used by some compilers and translators to simplify the derivation of the data dependence tests and the transformations used to discover parallelism. We show that loop normalization is the inverse of loop skewing for certain triangular loops, and can actually reduce the amount of parallelism available in certain cases.

Most methods for executing the loop:

```
do I = 2, N - 1
    do J = 2, N - 1
        A(I, J) = ...
    end do
end do
```

on a vector or parallel computer involve executing all iterations of the J loop in parallel or executing all the iterations of the I loop in parallel. Methods for detecting parallelism in one or the other of the *do* loops are well known, such as vectorization and loop interchanging.[1–6] However, if the loop contains the assignment:

```
do I = 2, N - 1
    do J = 2, N - 1
        A(I, J) = (A(I + 1, J) + A(I - 1, J) + A(I, J + 1)
                  + A(I, J - 1))/4
    end do
end do
```

then the loop is parallel for neither the J loop (due to the data dependence from $A(I, J)$ to $A(I, J-1)$) nor the I loop (due to the data dependence from $A(I, J)$ to $A(I-1, J)$). An alternate method to execute such loop nests in parallel, known as the *wavefront* method,[7–9] (or hyperplane method[10]) can be used to extract parallelism from loops such as this. Existing derivations of the wavefront method depend on finding the "angle" of the wavefront through the iteration space and optimizing this angle. (We ignore the 0° or 90° wavefront angles, which are really just the execution of a single loop in parallel.) The methods given to find the wavefront angle only work with very simple subscript expressions, and are not integrated with other well known data dependence tests for parallelism detection.

2. DATA DEPENDENCE AND LOOP INTERCHANGING

Data Dependence: Simple data dependence relations in scalar code are shown by the following program segment:

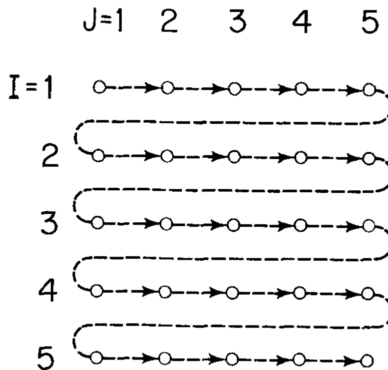
$$\begin{aligned} S_1&: \quad X = Z \\ S_2&: \quad Y = X \\ S_3&: \quad Z = Z + 1 \\ S_4&: \quad X = 9 \end{aligned}$$

Here we say that statement S_2 depends on S_1 *(flow-dependence)* since the value of X used in S_2 is assigned in S_1. We say that S_3 depends on S_1 *(anti-dependence)* since the value of Z assigned in S_3 is not the value used by S_1. Finally, we say that S_4 depends on S_1 *(output-dependence)* since the value assigned to X in S_3 is assigned after the value in S_1 is assigned. More on data dependence can be found in the references.[1,3,4,8,11–13]

Loop Interchanging: A nest of *do* loops, such as the two-nested loop in Fig. 1a, can be thought of as traversing the two-dimensional iteration space shown in Fig. 1b. The arrows in the iteration space represent how the serial *do* loops execute the statements in the loop for iteration $[I = 1, J = 1]$ first, then [1, 2], [1, 3],..., [1, 5], then incrementing the I index to go to [2, 1], [2, 2],..., [2, 5],..., [5, 1],..., [5, 5]. *Interchanging* these two *do* loops means changing the order in which the iteration space is traversed[1,2]; by

```
do I = 1,5
   do J = 1,5
      A(I,J+1) = A(I,J) + B(I,J)
   end do
end do
```

(a)



(b)

Fig. 1. (a) A two-nested *do* loop; (b) its iteration space.

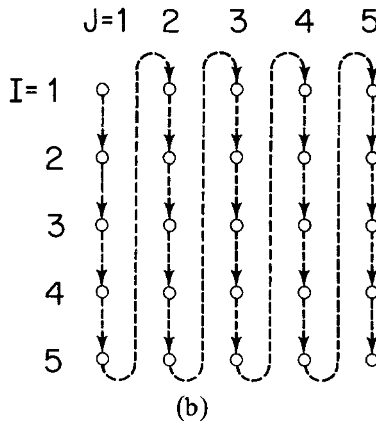
interchanging the loop as in Fig. 2a, the iteration space would be executed in the order shown in Fig. 2b.

Iteration Space Dependence Graph: For interchanging loops, a compiler is not so much interested in the data dependence relations between the statements in a loop as between the iterations of a loop. In these loops, the value of $A(1, 2)$ used in iteration $[I=1, J=2]$ was assigned in iteration $[I=1, J=1]$. In fact, the value used in iteration $[i, j]$ was assigned in iteration $[i, j-1]$ (except for boundary values); this relation is shown in the *iteration space dependence graph* Fig. 3. Notice that the pattern of the dependence flow in the iteration space can be characterized by the direction or the distance of the flow with respect to the loop dimensions. In this example, the dependence distance would be called $(0, -1)$, since the distance in the I dimension is zero, and the distance in the J dimension is -1. Many translators, such as the Parafrase Analyzer(5,6) and KAP,(14–16) save the sign of the dependence distance; here it would save $(0, -)$, or $(=, <)$, to characterize the data dependence in the loop ($(=, <)$ is the *data dependence direction vector*).

The possible directions in a two dimensional iteration space (corresponding to a doubly-nested do loop) are shown in Fig. 4. The iteration space dependence graph in Fig. 4a shows the data dependence directions that are preserved by loop interchanging. The one in Fig. 4b

```
do J = 1,5
   do I = 1,5
      A(I,J+1) = A(I,J) + B(I,J)
   end do
end do
```

(a)



(b)

Fig. 2. (a) The two-nested loop from figure 1 with the *do* loops interchanged; (b) the interchanged iteration space.

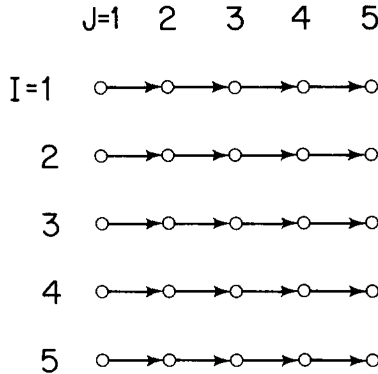


Fig. 3. The iteration space dependence graph for the loop in Figs. 1 and 2.

shows the data dependence directions that prevent loop interchanging; a pair of loops with a $(<, >)$ data dependence direction vector cannot be interchanged (without interaction from outer loops[1,2]).

Triangular loops: Some loops have non rectangular iteration spaces. Examples are loops with bounds that include outer loop indices, such as the loop nest in the following example. Here, the upper bound of the inner *do J* loop is a simple function of the outer loop index. The iteration space traversed by this *do* loop pair is drawn in Fig. 5; it is easy to see why this is called a triangular loop.

```
do I = 1, N
  do J = I, N
    A(I, J) = A(I, J) + B(I, J)
  end do
end do
```

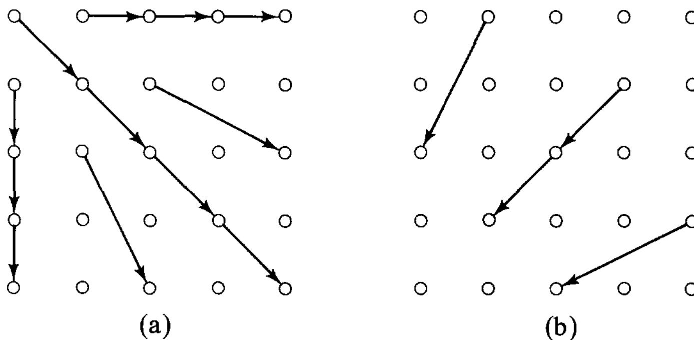


Fig. 4. (a) Dependence relations in the iteration space dependence graph that are preserved by loop interchanging. (b) Dependence relations that are violated by loop interchanging.

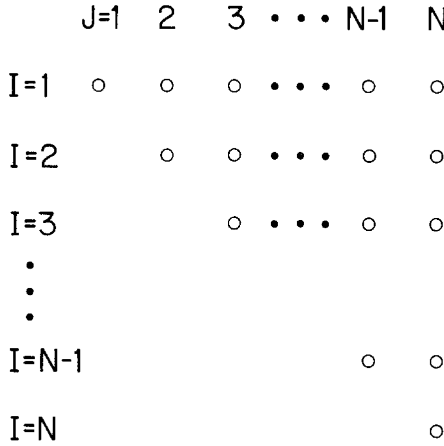


Fig. 5. Iteration space of a triangular loop.

To properly interchange these loops it is necessary to modify the loop bounds:

do $J = 1, N$
  do $I = 1, J$
    $A(I, J) = A(I, J) + B(I, J)$
  end do
end do

Triangular loop bounds as shown here actually appear quite often in numerical algorithms. A related class of loops, called *trapezoidal loops*, is similar in form to triangular loops, but is slightly more complicated to

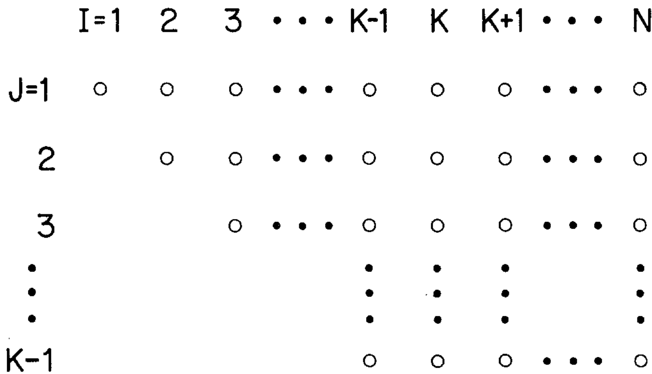


Fig. 6. Iteration space of a trapezoidal loop.

interchange. The next loop is a trapezoidal loop; the lower bound of the I loop is a function of the outer J loop, just as in a triangular loop, but the upper bounds do not match.

```
do J = 1, K - 1
   do I = J, N
      A(I, K) = A(I, K) + A(I, J) *A(J, K)
   end do
end do
```

The iteration space looks like a triangular loop with the lower point cut off at $K-1$ (the upper bound of the outer loop); see Fig. 6. To interchange these loops, a *min* function is needed to achieve the same cutoff point[17]:

```
do I = 1, N
   do J = 1, min(K - 1, I)
      A(I, K) = A(I, K) + A(I, J) * A(J, K)
   end do
end do
```

3. THE WAVEFRONT METHOD VIA INDEX SET SKEWING

Interchanging of trapezoidal loops can be the basis of a new formulation of the wavefront method of executing do loops.[7-10] The loop at the beginning of this article has the iteration space dependence graph as shown in Fig. 7. Even though the two *do* loops may be interchanged, neither loop may be executed in parallel. The wavefront method creates a "wave" that passes through the iteration space, as shown in Fig. 8. All the iterations on a single wavefront line are executed in parallel; this method exhibits much parallelism while still preserving all data dependence relations.

Our alternate formulation skews the index set of the original do loop creating a rhomboid iteration space out of what used to be a square; the modified iteration space dependence graph is shown in Fig. 9.

```
do I = 2, N - 1
   do J = I + 2, I + N - 1
      A(I, J - 1) = (A(I + 1, J - I) + A(I - 1, J - I)
                    +A(I, J + 1 - I) + A(I, J - 1 + I))/4
   end do
end do
```

This was done by adding I to the bounds of the inner J loop; notice that within the loop, J is replaced by the expression $J-I$ to account for the

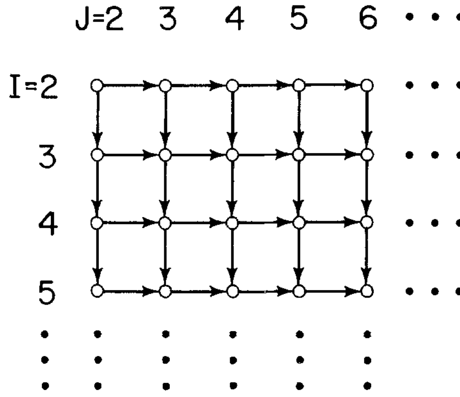


Fig. 7. Iteration space dependence graph of a loop that is a candidate for the wavefront method.

change in the bounds. The reader can verify that the first iteration of the new loop, $(I=2, J=I+2=4)$ still assigns to $A(2, 2)$. Note that the bounds for the J loop in the iteration space have changed slightly. Figure 9 also shows the iteration space dependence graph for the modified loop; notice that now not only are the loops interchangable, but when interchanged, the *do I* loop may be performed in parallel since there are no data dependence arcs which point straight down (with a direction vector of $(<, =)$). The *do*

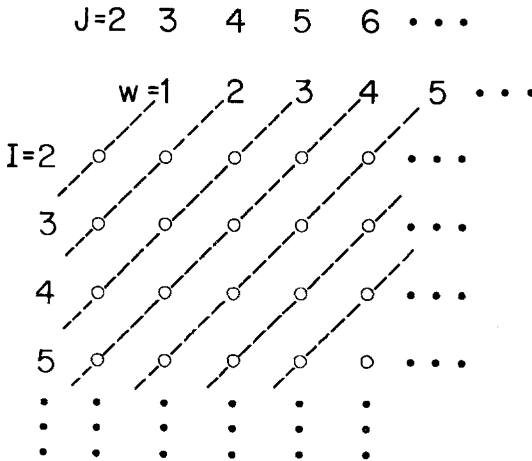


Fig. 8. Execution ordering for the wavefront method of executing two loops, superimposed on the iteration space dependence graph. All iterations on each diagonal line can be executed in parallel.

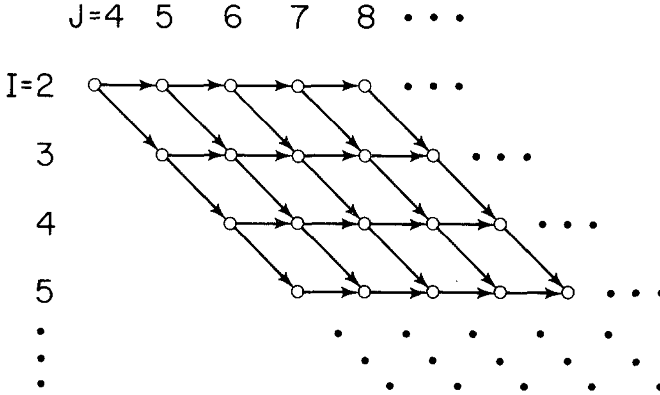


Fig. 9. Modified iteration space dependence graph after loop skewing.

loops may be interchanged using the techniques of trapezoidal loop interchanging; by executing the inner loop in parallel or vector mode, the wavefront ordering of the iterations will result:

```
do J=4, N+N-2
   do I=max(2, N-J+1), min(N-1, J-2)
      A(I, J-I)=(A(I+1, J-I)+A(I-1, J-I)
                   +A(I, J+1-I)+A(I, J-1+I))/4
   enddo
enddo
```

We can skew (index) J with respect to (outer loop index) I by (a factor of) f (where f is an integer constant) by

(a) replacing the lower bound of the J loop, LBJ, with the expression $(LBJ+I*f)$,

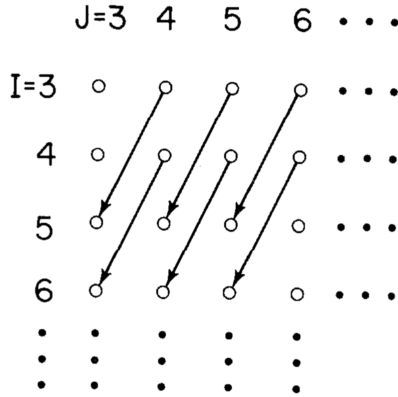
(b) replacing the upper bound of the J loop, UBJ, with the expression $(UBJ+I*f)$, and

(c) replacing all occurrences of J in the loop with the expression $(J-I*f)$.

Loop skewing is always legal; it has no effect on the numerical results of the program. It does not even change the execution order of the iterations (iterations in the skewed iteration space are executed in the same order as the corresponding iterations in the original iteration space). However, loop skewing does change the direction vectors. In this example, the "downward" dependences, with a direction of $(<, =)$, are changed to $(<, <)$ directions by loop skewing. This trick can even be used to change some $(<, >)$ direction vectors to $(<, <)$, as shown in Fig. 10. Also, a

```
do I = 3, N
   do J = 3, N
      A(I,J) = A(I-2,J+1) + ...
   end do
end do
```

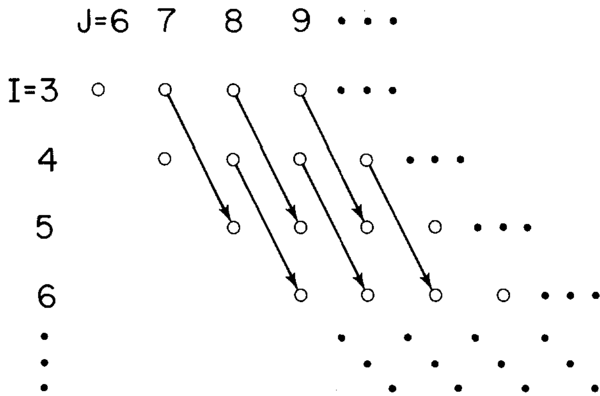
(a)



(b)

```
do I = 3, N
   do J = 3+I, N+I
      A(I,J-I) = A(I-2, J+1-I) + ...
   end do
end do
```

(c)



(d)

Fig. 10: Changing $(<, >)$ direction vectors to $(<, <)$ via loop skewing:

The loop in (a) has a data dependence relation with the direction vector $(<, >)$; the iteration space dependence graph for this loop is shown in (b). The $(<, >)$ dependence prevents the loops from being interchanged. If loop skewing is applied, as in (c), the iteration space dependence graph will be changed to (d); the $(<, >)$ direction vector has been modified to a $(<, <)$ direction vector, and the loops can be interchanged.

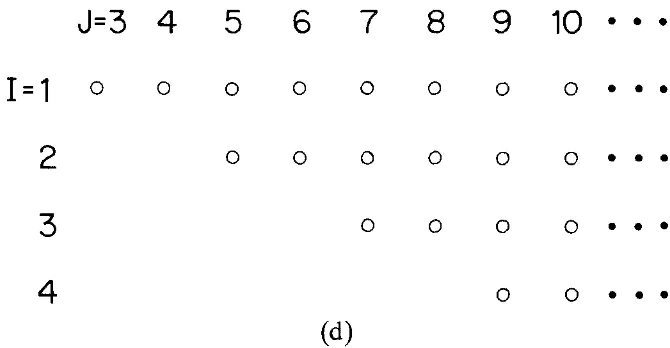
```
do I = 1, N
   do J = 1, N
      . . .
```

(a)

(b)

```
do I = 1, N
   do J = 1+2*I, N+2*I
      . . .
```

(c)



(d)

Fig. 11: Loop Skewing by a Larger Factor.

Loops can be skewed by a factor greater than one. The loop in (a) has the iteration space shown in (b). After skewing the inner loop by a factor of 2, the loop in (c) results. The new iteration space is shown in (d). The advantage of large factor skewing is that more $(<, >)$ data dependence direction vectors can be changed to $(<, <)$ directions, thus allowing loop interchanging and vectorization after skewing.

```
do I = 2, N-1
   do J = 2, M-1
      do K = 2, P-1
         X(I,J,K) = (X(I-1,J,K) + X(I+1,J,K)
                   + X(I,J-1,K) + X(I,J+1,K)
                   + X(I,J,K-1) + X(I,J,K+1)) / 6.
      end do
   end do
end do
```

(a)

```
do I = 2, N-1
   do J = 2, M-1
      do K = I+J+2, I+J+P-1
         X(I,J,K-I-J) = (X(I-1,J,K-I-J) + X(I+1,J,K-I-J)
                       + X(I,J-1,K-I-J) + X(I,J+1,K-I-J)
                       + X(I,J,K-I-J-1) + X(I,J,K-I-J+1)) / 6
      end do
   end do
end do
```

(b)

```
do K = 6, N+M+P-3
   do I = max(2,K-M-P), min(N-1,K-4)
      do J = max(2,K-I-P), min(M-1,K-I-2)
         X(I,J,K-I-J) = (X(I-1,J,K-I-J) + X(I+1,J,K-I-J)
                       + X(I,J-1,K-I-J) + X(I,J+1,K-I-J)
                       + X(I,J,K-I-J-1) + X(I,J,K-I-J+1)) / 6
      end do
   end do
end do
```

(c)

Fig. 12: Loop Skewing with respect to more than one loop.
The six point difference equation in (a) cannot be executed in parallel along any of the three *do* loop dimensions. We can skew the inner K loop index with respect to both the outer loop indices, as shown in (b). By interchanging this loop to the outermost nest level, as in (c), the inner two loops can both be executed in parallel.

loop can be skewed by a factor of 2 or more, as in Fig. 11. Finally, a loop can be skewed with respect to more than one loop, as shown in the example in Fig. 12.

4. LOOP NORMALIZATION

Loop normalization is a minor transformation performed by some other parallelism detection programs[3–6] in order to make data dependence testing easier. Loop normalization modifies the loop bounds so that the lower bound of all *do* loops is one (or zero), and the increment is one; this simplifies data dependence tests because two out of three of the loop bound expressions will be a simple known constant. The following example shows a *do* loop nest before and after loop normalization.

before:

```
do I = 3, N
  do J = I + 1, N
    A(I, J) = A(I - 1, J) + B(I, J)
  enddo
enddo
```

after:

```
do I = 1, N - 2
  do J = 1, N - I
    A(I + 2, J + I + 2) = A(I + 1, J + I + 2) + B(I + 2, J + I + 2)
  enddo
enddo
```

In this carefully concocted example, loop normalization may make data dependence equations easier to derive, but it also makes the job of a vectorizer more difficult. First, what used to be simple array subscripts ($A(I, J)$, $A(I-1, J)$) are now much more complicated, with two index variables in the second subscript ($A(I+2, J+I+2)$, $A(I+1, J+I+2)$). Second, and perhaps more important, the original loop exhibited a data dependence with a $(<, =)$ direction vector:

$[I=3, J=5]$ uses A (3, 5)
$[I=4, J=5]$ assigns A (3, 5)
$(-1, \ 0)$ data dependence distance vector
$(<, =)$ data dependence direction vector

Loop normalization has changed this to a $(<, >)$ direction vector:

$[I=1, J=2]$ uses A (3, 5)
$[I=2, J=1]$ assigns A (3, 5)
$(-1, +1)$ data dependence distance vector
$(<, >)$ data dependence direction vector

The $(<, >)$ direction vector prevents loop interchanging, if that is desirable for any reason. Loop normalization here is just a special case of index set skewing, with a negative skew factor. Because loop normalization can adversely affect the complexity of transforming the *do* loop nest, loop normalization should be avoided.

5. SUMMARY

Loop skewing is a simple transformation of the loop bounds that changes the shape of the iteration space. It also can change the data depen-

dence direction vectors to allow loop interchanging and to allow parallel or vector code to be generated after loop interchanging. Loop skewing is a convenient method for a compiler or translator to implement the wavefront method of executing a loop nest in parallel. Previous wavefront derivations have focussed on the dependence analysis needed to discover when wavefronting is useful. Loop skewing makes wavefronting easier to understand and implement. However, we are not introducing a new wavefront algorithm, only a simple vehicle to implement wavefronting.

Loop normalization is used by some translators to simplify the derivation and implementation of data dependence testing of other transformations that discover parallelism. In some cases it can be seen to be a form of loop skewing by a negative factor; in these cases, loop normalization will also change the data dependence direction vectors in such a way as to possibly adversely affect the types of transformations attempted by the translator. For this reason we do not recommend loop normalization for parallelizing translators.

REFERENCES

1. M. Wolfe, Optimizing Supercompilers for Supercomputers, Ph. D. Thesis, Dept. of Comp. Sci. Rpt. No. 82–1105, Univ. of Illinois, Urbana, Illinois, (October, 1982).
2. J. R. Allen and K. Kennedy, Automatic Loop Interchange, *Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, SIGPLAN Notices **19**, (6):233–246, (June 1984).
3. J. R. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, *Supercomputers: Design and Applications*, Kai Hwang, (ed.), IEEE Computer Society Press, Silver Spring, Maryland, pp. 186–203, (1982).
4. J. R. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, Tech. Rpt. COMP TR84–9, Rice Univ., Houston, Texas, (July 1984).
5. D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe, The Structure of an Advanced Vectorizer for Pipelined Processors, *Proc. of COMPSAC 80, The 4th Int'l Computer Software and Applications Conf.*, Chicaga, Illinois, pp. 709–715 (October 1980).
6. D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe, The Structure of an Advanced Retargetable Vectorizer, *Supercomputers: Design and Applications*, Kai Hwang (ed.), IEEE Computer Society Press, Silver Spring, Maryland, pp. 163–178 (1982).
7. Y. Muraoka, *Parallelism Exposure and Exploitation in Programs*, Ph. D. Thesis, Dept. of Comp. Sci. Rpt. No. 71–424, University of Illinois, Urbana, Illinois, (February 1971).
8. U. Banerjee, S. C. Chen, D. Kuck, and R. Towle, Time and Parallel Processor Bounds for Fortran-Like Loops, *IEEE Trans. on Computers*, **C–28** (9): 660–670 (September 1979).
9. R. H. Kuhn, Optimization and Interconnection Complexity for: Parallel Processors, Single—Stage Networks, and Decision Trees, Ph. D. Thesis, Dept. of Comp. Sci. Rpt. No. 80–1009, University of Illinois, Urbana, Illinois, (February 1980).
10. L. Lamport, The Parallel Execution of DO Loops, *Comm. of the ACM*, **17**(2):83–93 (February 1974).
11. U. Banerjee, Data Dependence in Ordinary Programs, M. S. Thesis, Dept. of Comp. Sci. Rpt. No. 76–837, Univ. of Illinois, Urbana, IL, (November 1976).

12. U. Banerjee, Speedup of Ordinary Programs, Ph. D. Thesis, Dept. of Comp. Sci. Rpt. No. 79–989, University of Illinois, Urbana Illinois, (October 1979).
13. D. Kuck, *The Structure of Computers and Computations*, Vol. I, John Wiley and Sons, Inc., New York, (1978).
14. C. Huson *et al.*, The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer, *Proc. of the 1986 Int'l Conf. on Parallel Processing*, St. Charles, IL, IEEE Computer Society Press, Washington, DC, pp. 827–832 (August 1986).
15. J. Davies *et al.*, The KAP/S–1: An Advanced Source—to—Source Vectorizer for the S–1 Mark IIa Supercomputer, *Proc. of the 1986 Int'l Conf. on Parallel Processing*, St. Charles, Illinois, IEEE Computer Society Press, Washington, DC, pp. 833–835 (August 1986).
16. T. Macke *et al.*, The KAP/ST–100: An Advanced Source—to—Source Vectorizer for the ST–100 Attached Processor, *Proc. of the 1986 Int'l Conf. on Parallel Processing*, St. Charles, Illinois, IEEE Computer Society Press, Washington, DC, pp. 171–175 (August 1986).
17. M. Wolfe, Advanced Loop Interchanging, *Proc. of the 1986 Int'l Conf. on Parallel Processing*, St. Charles, Illinois, IEEE Computer Society Press, Washington, DC, pp. 536–543 (August 1986).