Dependence Analysis of Arrays Subscripted by Index Arrays

Kathryn McKinley

CRPC-TR91163
July, 1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Dependence Analysis of Arrays Subscripted by Index Arrays [*]

Kathryn S. McKinley

Rice University
Department of Computer Science
P.O. Box 1892
Houston, TX 77251

kats@rice.edu

Abstract

When arrays are subscripted by index arrays, also known as subscripted subscripts, automatic vectorization and parallelization systems usually assume dependence or engineer run-time dependence testing. This work investigates the effectiveness of user assertions in determining dependence with static dependence analysis techniques. A set of basic user assertions about index variables are shown to be effective in eliminating dependences for general subscript expressions involving index variables, in many cases increasing the amount of parallelism available at compile time.

Keywords: dependence analysis, index arrays, subscripted subscripts, index variables, user assertions, parallelization, vectorization

1 Introduction

Index arrays are used frequently in numerical Fortran codes for a variety reasons. For example, they eliminate the need to swap values in the arrays subscripted by the index

[*]This research is supported by the National Science Foundation under grants ASC8518578 and CDA8619893, and by IBM.

array, which is common in codes such as Gaussian Elimination. In addition, they are used to eliminate computation when sparse systems are being solved. Two dependence testing studies have also found that index arrays occur and are a significant source of imprecision in dependence testing [SLY89, PP91].

When analyzing sequential Fortran for the purposes of parallelization or vectorization, a significant source of imprecision in dependence analysis results from the use of index arrays. Dependence analysis must be conservative in the sense that if it is possible for two references, where one is a write, to access the same memory location, then their original access order must be preserved [Ban79, Ban88, Wol89, AK87]. During static analysis it is usually impossible to determine possible values for index arrays, therefore dependence must be assumed for the arrays accessed by index arrays. For example, consider the following loop where index variables are referenced in the simplest possible way.

Loop A

```
              DO I = 1, N
      S1          A(INDEX(I)) = ···
      S2          ···          = A(INDEX(I))
              ENDDO
```

In loop A, loop-carried *true*, *anti*, and *output* dependences must be assumed between $S1$ and $S2$. These conservative dependences are necessary because the values of INDEX(I) are unknown and may overlap and intertwine. Most research and production compilers assume dependence for this reason.

In the literature, techniques which employ run-time dependence analysis [Pol86, Pol87] allow a general treatment of arrays accessed by index arrays. When index arrays contain random, patternless, or repeat values the conservative dependences determined statically really exist. If there is any parallelism to be extracted in this situation, run-time dependence testing is the only way to safely introduce it. Here, the overhead associated with run-time testing cannot be avoided. However, if the compiler or user determines that the index array in loop A is a permutation array, then there are no loop carried dependences on array A and the loop may be run in parallel without any run-time testing.

This work illustrates how a variety of user or compiler assertions about index arrays can be incorporated into dependence testing. We introduce a set of assertions and assume a framework in which the compiler or the user determines the assertions. Uses of index arrays are then examined in light of these assertions. We illustrate the increased precision of the dependence information possible with their use with a simple example. Then, the results are generalized for more complicated linear expressions involving index variables. We show that knowing the form of an index array enables the elimination of dependences, allowing a more precise dependence graph. This precision in turn may expose parallelism and other opportunities for optimization in programs.

2 Assertions

A selection of several common and observed properties of index variables is examined below. One frequent usage of index arrays of size n is as a permutation array of the integers 1 to n. In sparse matrix codes there also exist index arrays with size n, but whose values range to from 1 to k, where $k \geq n$. When each value in the index array is distinct, but the range is larger than n, it is called a *single valued* index array. Permutation arrays are simply a subset of single valued index arrays and are not treated separately here. Index arrays that increase or decrease corresponding to an increase or decrease in their subscript positions are also encountered in practice (see Section 5). The above array types form the framework for the assertions treated here and are enumerated formally below.

Single Valued	INDEX(I) $\neq$ INDEX(J), $I \neq J$
Monotone Non-Decreasing	INDEX(I) $\leq$ INDEX(I + 1)
Monotone Non-Increasing	INDEX(I) $\geq$ INDEX(I + 1)
Strictly Increasing	INDEX(I) $<$ INDEX(I + 1)
Strictly Decreasing	INDEX(I) $>$ INDEX(I + 1)

Strictly increasing is just a special case which combines single valued and monotone non-decreasing, and *strictly decreasing* is a combination of single valued and monotone non-increasing. An example of a single valued index array is INDEX(1:4) = {8, 1, 5, 2} and of a strictly increasing index array is INDEX(1:4) = {1, 2, 5, 8}.

3 A Simple Use of Index Arrays

In loop A, conservative dependence analysis must assume loop carried *true*, *anti*, and *output* dependence. There is also a loop independent *true* dependence, which exists regardless of the values of the index array. If the index array in loop A is single valued then there are no loop-carried dependences due to these references of array A. Clearly there is no dependence if the index array is strictly increasing or decreasing. However, if the index array is simply monotone non-decreasing or non-increasing, all possible dependences may be present.

Consider two more interesting, but simple uses of index variables in loops B and C.

```
Loop B
               DO I = 1, N
      S3           A(INDEX(I))  = ···
      S4           ···          = A(INDEX(I + 1))
               ENDDO
Loop C
               DO I = 2, N
      S5           A(INDEX(I))  = ···
      S6           ···          = A(INDEX(I - 1))
               ENDDO
```

In loop B, $S3$ and $S4$ have the following loop-carried dependences when there is no information about the values of INDEX(I).

S3 → S4	*true*	INDEX(I_1)	*may equal*	INDEX(I_2 + 1)	$I_1 \le I_2$
S4 → S3	*anti*	INDEX(I_2)	*may equal*	INDEX(I_1 + 1)	$I_1 \le I_2$
S3 → S4	*output*	INDEX(I_1)	*may equal*	INDEX(I_2)	$I_1 \neq I_2$

In loop B, $S5$ and $S6$ have the following dependences when there is no information about the values of INDEX(I).

S5 → S6	*true*	INDEX(I_1)	*may equal*	INDEX(I_2 - 1)	$I_1 \le I_2$
S6 → S5	*anti*	INDEX(I_2)	*may equal*	INDEX(I_1 - 1)	$I_1 \le I_2$
S5 → S5	*output*	INDEX(I_1)	*may equal*	INDEX(I_2)	$I_1 \neq I_2$

I_1 and I_2 are values of the loop index variable I on two different iterations of the loop. For the rest of this paper we assume I_1 occurs on an earlier iteration than I_2 ($I_1 < I_2$). Below we enumerate the assertions and their effects on this conservative dependence information for loops B and C. The conservative dependences that do not occur when an assertion is valid are indicated and a brief justification is given.

Single Valued

In loops B and C, *output* dependence does not occur because $\mathrm{INDEX}(I_1) \neq \mathrm{INDEX}(I_2)$.

Monotone Non-Decreasing

Loop B

True dependence does not occur.

Loop C

All dependences are still possible.

All dependences are still possible for loop C as can be observed in Example 1. However in loop B, a definition of A(INDEX(I)) at statement S3 is either killed by the next iteration or not referenced at all. Therefore, a carried *true* dependence is never instantiated.*

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\multicolumn{3}{c}{\textbf{Loop B}} & \multicolumn{3}{c}{\textbf{Loop C}} \\
A_1(1) & = & \cdots & A_2(1) & = & \cdots \\
\cdots & = & A_2(1) & \cdots & = & A_1(1) \\
A_2(1) & = & \cdots & A_3(1) & = & \cdots \\
\cdots & = & A_3(1) & \cdots & = & A_2(1) \\
A_3(1) & = & \cdots & A_4(2) & = & \cdots \\
\cdots & = & A_4(2) & \cdots & = & A_3(1) \\
A_4(2) & = & \cdots & A_5(3) & = & \cdots \\
\cdots & = & A_2(3) & \cdots & = & A_4(2)
\end{array}$$

Example 1: Let INDEX(I) = {1,1,1,2,3}

Monotone Non-Increasing

Loop B

True dependence does not occur.

Loop C

All dependences are still possible.

These follow from the monotone non-decreasing case.

*However, a simple reversal of statement S3 and S4 will instantiate the *true* dependence, indicating that in the general case it may be present.

Strictly Increasing

Loop B

Output dependence does not occur because $\mathrm{INDEX}(I_1) \neq \mathrm{INDEX}(I_2)$.

True dependence does not occur because $\mathrm{INDEX}(I_1) \neq \mathrm{INDEX}(I_2 + 1)$.

Loop C

Output dependence does not occur because $\mathrm{INDEX}(I_1) \neq \mathrm{INDEX}(I_2)$.

Anti dependence does not occur because $\mathrm{INDEX}(I_2) \neq \mathrm{INDEX}(I_1 + 1)$.

In Example 2, the *anti* dependence is instantiated in loop B and the *true* dependence in loop C. The other conservative dependences are proven not to exist using proof by contradiction. For loops B and C *output* dependence occurs only if $\mathrm{INDEX}(I_1) = \mathrm{INDEX}(I_2)$ where $I_1 < I_2$, but the definition of strictly increasing requires $\mathrm{INDEX}(I_1) < \mathrm{INDEX}(I_2)$, a contradiction. In loop B, *true* dependence occurs if $\mathrm{INDEX}(I_1) = \mathrm{INDEX}(I_2 + 1)$ where $I_1 < I_2$, but the definition of strictly increasing requires a stronger condition, i.e. $\mathrm{INDEX}(I_1) < \mathrm{INDEX}(I_2)$, a contradiction. Similarly, *anti* dependence in loop C is proven not to occur.

$$
\begin{array}{ccc@{\qquad\qquad}ccc}
\multicolumn{3}{c}{\textbf{Loop B}} & \multicolumn{3}{c}{\textbf{Loop C}} \\
A_1(1) & = & \cdots & A_2(2) & = & \cdots \\
\cdots & = & A_2(2) & \cdots & = & A_1(1) \\
A_2(2) & = & \cdots & A_3(3) & = & \cdots \\
\cdots & = & A_3(3) & \cdots & = & A_2(2) \\
A_3(3) & = & \cdots & A_4(4) & = & \cdots \\
\cdots & = & A_4(4) & \cdots & = & A_3(3)
\end{array}
$$

Example 2: Let INDEX(I) = {1,2,3,4,5}

Strictly Decreasing

Loop B

Output dependence does not occur because $\mathrm{INDEX}(I_1) \neq \mathrm{INDEX}(I_2)$.

true Dependence does not occur because $\mathrm{INDEX}(I_1) \neq \mathrm{INDEX}(I_2 + 1)$.

Loop C

Output dependence does not occur because $\mathrm{INDEX}(I_1) \neq \mathrm{INDEX}(I_2)$.

Anti dependence does not occur because $\mathrm{INDEX}(I_2) \neq \mathrm{INDEX}(I_1 + 1)$.

In Example 3, the *anti* dependence is instantiated in loop B and the *true* dependence in loop C. Using proof by contradiction, as with the strictly increasing case, the other dependences are proven not to occur.

Loop B			Loop C		
$A_1(5)$	$=$	$\cdots$	$A_2(4)$	$=$	$\cdots$
$\cdots$	$=$	$A_2(4)$	$\cdots$	$=$	$A_1(5)$
$A_2(4)$	$=$	$\cdots$	$A_3(3)$	$=$	$\cdots$
$\cdots$	$=$	$A_3(3)$	$\cdots$	$=$	$A_2(4)$
$A_3(3)$	$=$	$\cdots$	$A_4(2)$	$=$	$\cdots$
$\cdots$	$=$	$A_4(2)$	$\cdots$	$=$	$A_3(3)$

Example 3: Let INDEX(I) = {5,4,3,2,1}

4 General Uses of Index Arrays

4.1 Index Arrays subscripted by Linear Expressions

The subscript of an index variable may be a linear expression, as in loop D.

Loop D

```
              DO I = 1, N
S1               A(INDEX(αI ± γ)) = ···
S2               ···                 = A(INDEX(βI ± δ))
              ENDDO
```

For simplicity assume α, β, γ, and δ are positive integers. Loop D is similar to loops A, B and C in that *true*, *anti*, and *output* loop-carried dependence must be assumed if the values of the index array is unknown. This form can be examined by applying existing dependence tests [Ban88, Wol89, GKT91] to the subscripts of the index variables, and then applying the results to the array references referenced by the index variables. Determining equality of $\alpha I \pm \gamma$ and $\beta I \pm \delta$ can be done using the single index variable (SIV) exact test [GKT91]. If this test proves independence, then the same dependences that are eliminated in Section 3 can be eliminated based on relationship between $\alpha I \pm \gamma$ and $\beta I \pm \delta$. However, no additional dependences may be eliminated.

Below each assertion is listed with the method to determine if a conservative dependence can be deleted, given an index variable of the form in loop D.

Single Valued

The *output* dependence ($S1 \rightarrow S1$) does not occur.

Monotone Non-Decreasing and Monotone Non-Increasing

If (1) $\alpha I \ \pm \ \gamma$ *is always* $\leq \beta I \ \pm \ \delta$, and (2) $\alpha I_1 \ \pm \ = \gamma \ \beta I_2 \ \pm \ \delta$ for some $I_1 \ \leq \ I_2$, then *true* dependence does not occur. We use the SIV exact test determine condition (2). The direction vector resulting from this test indicates if condition (1) holds. If condition (1) holds, then the *true* dependence may be eliminated.

Strictly Increasing and Strictly Decreasing

The *output* dependence ($S1 \rightarrow S1$) does not occur.
Knowing that all the values of index will be unique and increasing, we only need to determine if $\alpha I \ \pm \ \gamma$ and $\beta I \ \pm \ \delta$ may be equal. This equality can also be determined using the single index variable (SIV) exact test. If the test determines they are never equal, then no *true, anti, or output* dependence exists. If a dependence exists, the SIV test reports a direction vector that may eliminate either the *anti* or *true* dependence.

4.2 Linear Expressions using Index Arrays

Index arrays may also be used in linear expressions to subscript other arrays. Consider the form of the subscripts in loop E. When this form occurs, the assertions also allow additional dependences to be deleted. However, if γ or δ are functions of an index variable as well, more general tests are required (Section 5 illustrates a program of this form). For example, the multiple induction variable (MIV) test [GKT91] can be applied and the resulting direction vector used to eliminate dependences.

Below, we consider the two specific cases in loops F and G where $\delta \ > \ 0$. The dependences determined in these specific cases are at best what the general case may delete. For the general case, the direction vector from the SIV or MIV test can be used to eliminate the dependences outlined below for these specific cases.

Again, without assertions all three types of dependences must be assumed. With assertions it is possible to increase the precision of the dependence information. The increases possible in precision are outlined below. Remember $I_1 < I_2$.

Loop E

```
          DO I = 1, N
  S1          A(α INDEX(I) ± γ)) = ···
  S2          ···                 = A(β INDEX(I) ± δ))
          ENDDO
```

Loop F

```
          DO I = 1, N
  S1          A(INDEX(I)) = ···
  S2          ···          = A(INDEX(I) + δ))
          ENDDO
```

Loop G

```
          DO I = 1, N
  S3          A(INDEX(I)) = ···
  S4          ···          = A(INDEX(I) − δ))
          ENDDO
```

Single Valued

Loop F

Output dependence does not occur because $\mathrm{INDEX}(\mathrm{I}_1) \neq \mathrm{INDEX}(\mathrm{I}_2)$.

Loop G

Output dependence does not occur because $\mathrm{INDEX}(\mathrm{I}_1) \neq \mathrm{INDEX}(\mathrm{I}_2)$.

Monotone Non-Decreasing

Loop F

True dependence does not occur because $\mathrm{INDEX}(I_1) < \mathrm{INDEX}(I_2) + \delta$.
Note, $\mathrm{INDEX}(I_1) \leq \mathrm{INDEX}(I_2)$.

Loop G

Anti dependence does not occur because $\mathrm{INDEX}(I_1) > \mathrm{INDEX}(I_2) - \delta$.
Note, $\mathrm{INDEX}(I_1)$ *is already* $\geq \mathrm{INDEX}(I_2)$.

In Example 4 a monotone non-decreasing example is given for loops F and G where $\delta = 1$. The subscript of A is the value of I, and the parentheses contain the appropriate value of either INDEX(I), INDEX(I) + 1, or INDEX(I) - 1 that corresponds with I.

Monotone Non-Increasing

Loop F

Anti dependence does not occur because $\mathrm{INDEX}(I_1) < \mathrm{INDEX}(I_2) + \delta$.
Note $\mathrm{INDEX}(I_1) \leq \mathrm{INDEX}(I_2)$.

Loop G

True dependence does not occur because INDEX(I_1) > INDEX(I_2) $- \delta$.

In Example 5 a monotone non-increasing example is given for loops E and F where $\delta = 1$. The subscript of A is the value of I, and the parentheses contain the appropriate value of either INDEX(I), INDEX(I) + 1, or INDEX(I) - 1 that corresponds with I.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\multicolumn{3}{c}{\text{Loop E}} & \multicolumn{3}{c}{\text{Loop F}} \\
A_1(2) & = & \cdots & A_1(2) & = & \cdots \\
\cdots & = & A_1(3) & \cdots & = & A_1(1) \\
A_2(2) & = & \cdots & A_2(2) & = & \cdots \\
\cdots & = & A_2(3) & \cdots & = & A_2(1) \\
A_3(2) & = & \cdots & A_3(2) & = & \cdots \\
\cdots & = & A_3(3) & \cdots & = & A_3(1) \\
A_4(3) & = & \cdots & A_4(3) & = & \cdots \\
\cdots & = & A_4(4) & \cdots & = & A_4(2) \\
A_5(3) & = & \cdots & A_5(3) & = & \cdots \\
\cdots & = & A_5(4) & \cdots & = & A_5(2) \\
A_6(4) & = & \cdots & A_6(4) & = & \cdots \\
\cdots & = & A_6(5) & \cdots & = & A_6(3) \\
A_7(5) & = & \cdots & A_7(5) & = & \cdots \\
\cdots & = & A_7(6) & \cdots & = & A_7(4)
\end{array}$$

Example 4: Let INDEX(I) = {2,2,2,3,3,4,5}, and $\delta = 1$

Strictly Increasing

Loop F

Output dependence does not occur.

True dependence (S1 $\longrightarrow$ S2) does not occur.

Loop G

Output dependence does not occur.

Anti dependence (S4 $\longrightarrow$ S3) does not occur.

The results from the monotone non-decreasing assertion are clearly still applicable. Combining this with the single value assertion, eliminates two of the three dependences for each loop.

Loop E	Loop F
$A_1(5) = \cdots$	$A_1(5) = \cdots$
$\cdots = A_1(6)$	$\cdots = A_1(4)$
$A_2(5) = \cdots$	$A_2(5) = \cdots$
$\cdots = A_2(6)$	$\cdots = A_2(4)$
$A_3(5) = \cdots$	$A_3(5) = \cdots$
$\cdots = A_3(6)$	$\cdots = A_3(4)$
$A_4(4) = \cdots$	$A_4(4) = \cdots$
$\cdots = A_4(5)$	$\cdots = A_4(3)$
$A_5(3) = \cdots$	$A_5(3) = \cdots$
$\cdots = A_5(3)$	$\cdots = A_5(2)$
$A_6(2) = \cdots$	$A_6(2) = \cdots$
$\cdots = A_6(2)$	$\cdots = A_6(1)$

Example 5: Let INDEX(I) = {5,5,5,4,3,2}, and $\delta = 1$

Strictly Decreasing

Loop F

Output dependence does not occur.

Anti dependence (S2 $\rightarrow$ S1) does not occur.

Loop G

Output dependence does not occur.

True dependence (S3 $\rightarrow$ S4) does not occur.

The results from the monotone non-increasing assertion are still applicable and combining this with the single value assertion, again eliminates two of the three dependences for each loop.

5 Index Arrays in Practice

We have several Fortran programs that contain simple uses of single valued index arrays as in loop A. However, we have also found programs with more interesting uses of index arrays. Below we discuss two programs that use index arrays. The first program uses its index array in a straight forward manner. The second uses its index array in an expression of the form discussed in Section 4.2. The values of both index arrays are regular patterns and strictly increasing.

MDG

In the Perfect benchmark MDG, a molecular dynamics program for the simulation of liquid water, the main routine assigns the index array IX(I) as follows.

```
              II = 1
              DO 100 I = 1, NVAR
                     IX(I) = II
100                  II = II + NATMO
```

The sample input provided with this program assign NVAR = 24 and NATMO = 1029. This loop results in IX(I) = NATMO * (I - 1) + 1, for I = 1, 24, a regular, strictly increasing expression. With advanced symbolic propagation and the problem size parameters a compiler could determine this expression. VAR(IX(C)), where C is a constant, is then passed into subroutines and used in non-trivial ways. In the subroutine initia the following code is found

```
          IJ = 1
          DO 2000 I = 1, NMOL
            DO 100 I = 1, NATOMS
              VX(IJ) = ···
              VY(IJ) = ···
              VZ(IJ) = ···
              ···
              IJ = IJ + 1
2000      CONTINUE
```

where the formal parameter VX is bound to VAR(IX(4)), VY to VAR(IX(5)), and VZ to VAR(IX(6)). In addition, NMOL = 3, NATOMS = 393, and NATMO = 3 * 393 = 1029. The access to each VX, VY, and VZ are therefore unique. Because the Fortran standard guarantees no aliasing of actual parameters, most dependence tests assume independence here and either of the loops may be made parallel. It is interesting to note that many compilers apply procedure inlining to enable dependence testing in the presence of procedure calls and in hopes of improving the precision of dependence testing. In this case, the opposite is true. Without index array information, inlining here causes the loss of the no aliasing assumption and requiring that *output* dependence be assumed.

TRFD

Below in the Perfect benchmark TRFD, a missile tracking program, the main routine assigns the index array IA(I). It is subsequently used via a subroutine call to TRFPRT. TX is an array of dimension 1000000.

```
      DO 30 NUM = 10, 40, 5
        DO 10 I = 1, NUM
10        IA(I) = (I * (I - 1)) / 2
        NORB = NUM
        NP = NUM
        NPQ = (NUM * (NUM + 1)) / 2
        ...
        IF (OUT) CALL TRFPRT(NP, NPQ, IA, X, ··· )
30      CONTINUE

SUBROUTINE TRFPRT(NORB, NIJ, IA, TX, ··· )
      ...
      DO 140 I = 1, NORB
        DO 130 J = 1, I
          DO 120 K = 1, I
            LMAX = K
            IF (K .EQ. I) LMAX = J
            DO 110 L = 1, LMAX
              IJ = IA(I) + J
              KL = IA(K) + L
              VAL = TX(IJ,IK)
        ...
```

In this case, the usage of index array IA in an expression is slightly disguised and there is no subsequent write of TX. However, this program illustrates that index arrays are being used in linear expression to subscript other arrays.

6 Discussion

In order to further validate the usefulness of assertions in the parallelization process several other issues must also be addressed. Currently in Fortran and other languages, there are no syntactic constructs that allow users to make assertions. Some systems provide compiler directives which usually are embedded in comments. Directives have several drawbacks. One is scoping the assertions. Does the assertion apply for the entire program, subroutine, or loop? As Polychronopoulos points out, occasionally index variables are defined in the same loop as they are used, further complicating the assertion process [Pol86].

In the ParaScope Editor [BKK$^+$89, KMT91, KMT90], an interactive parallel programming tool we are exploring an assertion facility with a language of its own, which would annotate programs and their dependence graphs. Assertions in this tool may take many forms. They may be applied to specific references, or a range of references (such as a loop or subroutine), or something as complex as all the uses of a particular variable. A facility like this would allows users to avoid the tedium of having to insert an assertion before every use of a index variable. We believe that semantic assertions about program facts will be an improvement over current compiler directives such as "run this loop in parallel regardless of the dependences", which communicates none of a users reasoning. ParaScope is already providing program dependence information to the user, and we hope that providing a more flexible format for user and compiler communication will further improve the parallelization process.

We are also exploring more sophisticated compiler techniques for symbolic propagation of program information such as index array properties [KMT91]. For example, if an index variable is determined to be single valued, how can this be propagated to other loops or procedures? In addition, detecting permutation arrays may require constant propagation into arrays, a difficult problem. Eventually it may be possible for a compiler to determine properties of index arrays. For example, the programs in Section 5 lead us to believe some index arrays could be treated as "shadow loops", where each index array reference is viewed merely as another loop induction variable whose lower bound, upper bound, and step size are determined using symbolic propagation.

Theoretically, the compiler solution is ideal because user assertions may introduce errors into correct programs. Practically, the symbolic analysis needed to support the compiler solution in not yet precise or efficient enough for general use.

7 Conclusions

This work illustrates that in practice, index arrays may have regular and discernible forms and uses. In addition this work shows that user assertions or advanced symbolic compiler analysis about index arrays may be used to increase the precision of dependence testing, thereby increasing opportunities for parallelization and optimization. However, the symbolic techniques needed to determine the form of an index array are not yet available in a practical compilation system.

References

[AK87] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[Ban79] U. Banerjee. *Speedup of ordinary programs.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1979. Report No. 79-989.

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

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[GKT91] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[KMT90] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. Technical Report TR90-137, Dept. of Computer Science, Rice University, October 1990. To appear in *IEEE Transactions on Parallel and Distributed Systems.*

[KMT91] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[Pol86] C. Polychronopoulos. *On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, August 1986.

[Pol87] C. Polychronopoulos. Advanced loop optimizations for parallel computers. In *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June 1987. Springer-Verlag.

[PP91] P. Petersen and D. Padua. Experimental evaluation of some data dependence tests. Technical Report 1080, CSRD, University of Illinois at Urbana-Champaign, February 1991.

[SLY89] Z. Shen, Z. Li, and P. Yew. An empirical study on array subscripts and data dependences. In *Proceedings of the 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.