From the Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96), Sunnyvale, CA, August 1996

Dependence Analysis of Fortran90 Array Syntax

Gerald Roth Ken Kennedy
Department of Computer Science
Rice University
Houston, Texas, U.S.A.

Abstract

Dependence analysis and dependence information are critical components of many optimizing and parallelizing compilers. And there exist many fast and precise dependence tests that work on scalar-subscripted array references. We have extended this analysis by adding tests that directly handle Fortran 90 array-section references. This paper describes our testing methodology and how we have extended direction vectors to contain the additional information.

Keywords: dependence analysis, Fortran90, array syntax, direction vectors, scalarization

1 Introduction

Fortran 90 and High-Performance Fortran (HPF)[9] are increasingly becoming the language of choice for creating high-performance applications targeted for today's high-end architectures, no matter whether those architectures are parallel, vector or superscalar. The array constructs of these languages have raised the level of abstraction from the strictly scalar constructs of Fortran 77, thus making them more expressive. Unfortunately, few compilers have taken advantage of this heightened level of abstraction. Instead they prefer to scalarize the array constructs into familiar scalar constructs which they then optimized by standard analyses and transformations.

In this paper we present a methodology for performing data dependence analysis directly on Fortran 90 array-section references. We show how direction vectors can be extended to include the dependence information. We also introduce a special class of dependences that arise from array syntax, and discuss some of their properties. Finally, we show how this additional information can be used to perform advanced program transformations.

1.1 Fortran 90

It is assumed that the reader is familiar with Fortran 90 [1], especially with the execution semantics of array operations. In Fortran 90, operations deal with their operands as unitary objects, even when they are arrays. Thus all right-hand side elements of an

array assignment statement are read before any left-hand side elements are stored. Arrays or subsections of arrays can be specified by using *triplet* subscripts. A triplet specifies a range in the form `[lower bound]:[upper bound][:stride]`. If the lower or upper bounds are not specified, the declared bounds of the array are assumed. The stride is 1 if not given.

Before we begin we'd like to clarify some terminology that is used in this paper. An *array reference* is a subscripted variable reference. A *subscript* is one element from a subscript list. A triplet, as defined above, is one type of subscript. It is assumed that whole array references, array references without a subscript list, have been modified within the compiler's internal representation to have a subscript list containing the appropriate number of null triplets.

1.2 Data Dependence

The theory of *data dependence* is well understood and is extensively used in advanced optimizing and parallelizing compilers. We say that a data dependence exists between two statements if there is an execution path from one to the other and both statements access the same memory location. Data dependence is fundamental to compilers that attempt reordering transformations since it specifies statement orderings that *must* be preserved to maintain program semantics [2, 15, 17].

There are four types of data dependence. *True dependence* occurs when one statement writes a memory location that another statement later reads. *Antidependence* occurs when one statement reads a memory location that another statement later writes. *Output dependence* occurs when one statement writes a memory location that another statement later writes. *Input dependence* occurs when one statement reads a memory location that another statement later reads.

Dependence testing is the process of determining whether a data dependence exists between two statements [5]. Dependence testing is mostly concerned with determining dependences that arise from subscripted array references that appear within loop nests, since it is not always easy to determine if such references access the same memory location.

1.3 Partition-based Dependence Testing

In the partition-based dependence testing algorithm [8] used in the analysis and transformation systems at Rice University, pairs of array references are classified before being tested. This allows us to chose the most efficient test for a given pair of references and lets us test the subscripts in the order of less expensive to more expensive. The classification system consists of two orthogonal criteria: *complexity* and *separability*.

Complexity refers to the number of distinct loop induction variables that appear within a subscript. Individual subscripts are first classified, and then those results are used during dependence testing to derive a classification for a subscript pair. Current complexity classes include ZIV (zero index variables), SIV (single index variable), and MIV (multiple index variables).

Separability refers to whether or not different subscript positions contain common induction variables. A subscript position is *separable* if the indices it contains do not

1. Partition the subscripts into separable and minimal coupled groups.
2. Label each subscript pair as ZIV, SIV, or MIV.
3. For each separable subscript pair, apply the appropriate single subscript test based upon the complexity of the subscripts.
4. For each coupled group, apply a multiple script test.
5. If any test yields independence, no dependences exist.
6. Otherwise merge all the direction vectors computed by the previous steps into a single set of direction vectors for the two references.

Figure 1: Partition-based dependence testing algorithm.

appear in other subscript positions [2, 7]. If different subscript positions contain the same index, they are said to be *coupled* [12]. The concept of separability is important when testing multidimensional arrays in that it allows dependence testing to proceed subscript-by-subscript without a loss of precision. In contrast, coupled subscripts must be tested as a group to obtain exact results.

The concepts of complexity and separability are combined in the partition-based dependence testing scheme to determine the most appropriate test to use for a given pair of references. An outline of the algorithm is given in Figure 1. This algorithm has been used with great success in the PFC compiler [3], the ParaScope programming environment [11], and the Fortran D compiler [10, 14].

2 Dependence Representation

Data dependences are often represented using *direction vectors* and/or *distance vectors* [15]. These vectors are convenient methods for characterizing the relationship between the values of the loop indices of the two array references involved in the dependence. In this paper we will only discuss direction vectors, although the algorithms presented could easily be made to work with distance vectors.

Direction vectors are useful in determining if a dependence is *loop-carried* or *loop-independent* [3]. For loop-carried dependences, the direction vector also tells us which loop *carries* the dependence and in which direction. The vectors contain an element for each loop which encloses both statements involved in the dependence. The positions in the vectors from left to right correspond to the surrounding loop indices from outermost to innermost.

2.1 Dependence Vectors and Array Sections

To extend direction vectors for array-section references, we add vector elements to account for the implied loops of the triplets. The number of elements added to a vector corresponds to the number of triplets that the two array references have in common.

```
    DO I = 1, N-1
S1:   A(I,2:N-1,1:N) = A(I,1:N-2,1:N) + A(I,2:N-1,1:N)
S2:   B(I,2:N-1,1:N) = A(I,3:N,1:N) + A(I+1,2:N-1,1:N)
    END DO
```

Figure 2: Fortran 90 code fragment.

In most cases these vector elements will only be considered when the two references originate from the same statement, in which case they will have the same number of triplets. These new direction vector elements will appear to the right of those elements corresponding to surrounding loops. We will order the elements from left to right as they appear in the subscript list, although any consistent ordering will do. In fact some people may want to use the opposite ordering since they want the rightmost direction vector position, corresponding to the innermost loop, to be associated with the leftmost subscript due to the column-major storage layout of Fortran arrays. We chose the left to right ordering for its ease of understanding since it matches the order in which the triplets appear in the program text.

Consider the code fragment shown in Figure 2. Any dependences among statements S_1 and S_2 due to the references to array A would have an associated direction vector containing three elements: the first corresponding to the I loop, the second corresponding to the first triplet, and the third corresponding to the second triplet.

2.2 Scalarization Dependences

Given this extension to the concept of a direction vector, there is a subclass of dependences that deserve some special attention: those dependences which have an "=" in all non-triplet direction vector positions. We call these dependences *scalarization dependences*. Since scalarization dependences arise from parallel constructs in the Fortran 90 program, they do not have the same behavior as non-parallel dependences. Note that it is valid for *any* of the three direction specifiers to appear in the triplet-related vector positions. Thus for scalarization dependences, it is no longer the case that a true dependence with a ">" as the first non-"=" direction is equivalent to an antidependence with the direction reversed, as has been previously noted [4, 6].

By definition, scalarization dependences are loop-independent with regard to surrounding loops. This has several implications. First, any such dependence of a statement on itself will always be an antidependence (ignoring input dependences), whereas such a dependence from one statement to a subsequent one will either be a true or output dependence. Next, scalarization dependences have no effect on the parallelization of surrounding loops, regardless of what direction the triplet-related positions contain. Finally, it is especially important to point out that such dependences do not affect the ability to parallelize the DO-loops that get generated during the scalarization of the Fortran 90 code. This is due to the fact that the array-section subscripts are explicitly parallel constructs.

But this does not mean that we can ignore scalarization dependences. These dependences will play an important role when the compiler scalarizes the Fortran 90 program into its Fortran 77 equivalent. This aspect of the dependences will be addressed in more

```
      DO J = 1, N
S3:     A(J:N,K,1:N) = A(J:N,1:N,L) + ···
      END DO
```

Figure 3: Complexity and separability example.

detail later in the paper.

As an example, we will once again refer to the code in Figure 2. This fragment of code contains the following scalarization dependences: $S_1\overline{\delta}_{(=,>,=)}S_1$, $S_1\overline{\delta}_{(=,=,=)}S_1$, and $S_1\delta_{(=,>,=)}S_2$. The code also has the dependence $S_2\overline{\delta}_{(<,=,=)}S_1$ which is carried by the I loop.

3 Classification of Array-Section References

As introduced in Section 1.3, pairs of array references are classified before being tested by the partition-based dependence testing algorithm. This allows us to chose the most efficient test for a given pair of references. We now explain how we have extended the concepts of complexity and separability to include array-section references.

3.1 Complexity

We have created a new complexity class for subscripts containing array syntax. We call this new class simply TRIPLET, corresponding to the triplet notation used in the subscript. Unlike the other complexity classes, a TRIPLET is also subclassified to indicate the complexity of its components. A TRIPLET is subclassified as SIV if the corresponding triplet subscript contains no index variables (the SIV subclassification is due to the index variable implicit in the triplet notation). If the triplet contains one or more index variables from enclosing loops in any of its components, then the TRIPLET is subclassified as MIV. When convenient we will use the shorthand TRIPLET_{SIV} and TRIPLET_{MIV} to represent the complexity and subclassification of a triplet subscript.

Statement S_3 in Figure 3 has two array references, each containing two subscripts that are classified as TRIPLET. For each reference, the first triplet subscript is subclassified as MIV due to the induction variable J, and the second triplet subscript is subclassified as SIV.

3.2 Separability

The concept of separability is an important issue for a dependence testing algorithm that is interested in both precision and efficiency. It allows dependence testing to proceed subscript-by-subscript, thus breaking down the problem space into smaller pieces, without a loss of precision. Luckily, array-syntax subscripts can cause different subscript positions to become coupled in only one situation: if corresponding triplets for the two array references being tested are in different subscript positions, those positions must be coupled.

Considering again the two references to array A in Figure 3, we see that the second and third subscript positions become coupled since the second triplet appears in each of them. The first subscript position is separable. Note however, that if the induction variable J appeared in another subscript position, that position would be coupled with the position that contains the first triplet. That coupling would be due to J though and has nothing to do with the triplet.

4 Dependence Testing of Array Expressions

We have designed our dependence testing methodology to fit into the partition-based testing scheme presented in Figure 1. The algorithm is first modified to include the determination of separability and complexity for triplet subscripts as defined above. In the following subsections we introduce the necessary decision algorithms that will determine independence of array-section references, or determine dependence and produce the desired direction vectors. As with most dependence testing schemes, we assume that all expressions used in the subscripts and triplets are linear in the loop induction variables. If nonlinear expressions are encountered we assume all direction vectors are possible.

In some situations that may occur, we will be exploiting existing SIV and MIV tests. This will necessitate the generation of the appropriate input parameters. To accomplish this we translate the triplet notation into a linear function of a pseudo-induction variable i. The pseudo-induction variable will run from 1 to $(ub - lb + st)/st$, where lb, ub, and st are the lower bound, upper bound, and stride of the triplet, respectively. The linear function that will be used in place of the triplet within the dependence tests is then $st * i + (lb - st)$. When the lower bound and stride are both one, this simplifies to i. This translation does not need to be applied to the program representation; it is only needed to produce the necessary input parameters for existing tests when required.

4.1 Separable Triplet Subscript Tests

The dependence test used for a separable subscript pair in which one of the subscripts is a triplet will depend upon the triplet's subclassification as well as the complexity of the other subscript. If the triplet is classified as TRIPLET_{SIV} and the other subscript is ZIV, the pair has the form $\langle lb : ub : st, c_1 \rangle$. For this situation, we define the dependence distance to be:

$$d = \frac{c_1 - lb + st}{st} \tag{1}$$

A dependence exists if and only if d is an integer and is in the range $1 : (ub - lb + st)/st$. For the common case where both lb and st are equal to one, d is simply equal to c_1. The dependence direction is calculated in the normal fashion by comparing d to zero.

Next we'll consider when both subscripts in the pair to be tested are classified as TRIPLET_{SIV}. In this case an SIV test is required, and we can exploit the existing SIV test implemented in the system. To use the test we must produce the appropriate input parameters by generating the linear functions of a pseudo-induction variable, as explained above.

However, there are two common cases which can be tested quite easily without

requiring the conversion.The first case is when both triplets have a stride of one. In that situation the dependence distance is simply the difference of the lower bounds:

$$d = lb_1 - lb_2 \tag{2}$$

A dependence exists if and only if $|d| \leq ub_1 - lb_1$. The second case is when both triplets have the same non-unit stride, in which the dependence distance is:

$$d = \frac{lb_1 - lb_2}{st_1} \tag{3}$$

In this case a dependence exists if and only if d is an integer and $|d| < (ub_1 - lb_1 + st_1)/st_1$. Most triplet subscripts are expected to fall into one of these two special cases, and as can be see in Equations 2 and 3, the dependence tests for these cases are simple and easy to compute.

In all other cases of separable subscript pairs that contain at least one triplet an MIV test will be required. We will rely upon the existing MIV tests implemented in the system in the same manner as we used the SIV test above, by converting the triplet into a linear function of a pseudo-induction variable.

4.2 Coupled Triplet Subscript Test

When triplet subscripts are coupled with other subscripts we will need to exploit a multi-subscript test. Again we will utilize the existing testing algorithms available in the system by converting the triplets into linear functions of pseudo-induction variables. However, this conversion has a special consideration in the case of coupled subscripts. If the subscripts became coupled because corresponding triplets did not appear in matching subscript positions, then the linear functions generated for the corresponding triplets will share the same pseudo-induction variable. Once the triplets have been translated, we can exploit whichever multi-subscript test is available in our system [12, 13, 16].

5 Advanced Scalarization

Before an array statement can executed on the target architecture, it must be rewritten so that it accesses smaller chunks of data. The size of the chunks must "fit" the hardware of the target machine, whether that be individual array elements for scalar machines or array sections for vector machines. This translation is known as *scalarization* when discussed in terms of a scalar machine, and is called *sectioning* on vector architectures. In this section we address compiling to a scalar machine, although the material is equally applicable to vector machines.

5.1 Two-Pass Scalarization

Due to the semantics of array statements, their translation into correct serial code is not always trivial, as can be seen in Figure 4. The code in Figure 4b, the result of a naive scalarization, is not equivalent to its Fortran 90 counterpart since on the second and subsequent iterations of the I loop the reference $A(I-1)$ will access the new values of the array A assigned on the previous iteration.

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

(a) array statement

```
DO I=2, N
  A(I) = A(I-1) + B(I-1)
END DO
```

(b) naively scalarized code

Figure 4: Invalid scalarization example.

Fortunately, data dependence information can tell us when the scalarized loop is correct. Allen and Kennedy [4] have shown that a scalarized loop is correct if and only if it does not carry a true dependence. Using this fact, most compilers perform scalarization in the following manner:

1. Perform a naive scalarization of the array statement into scalar code.
2. Compute the data dependences of the resulting code.
3. While a loop carries a true dependence, perform code transformations to either eliminate the dependence or change it into an antidependence.

The code transformations that can be applied to handle the loop carried true dependences include loop reversal, loop interchange, prefetching, and as a last resort the generation of array temporaries. The interested reader is referred to Allen and Kennedy [4] for a complete discussion.

Note that the above algorithm requires two passes over the code, one to perform the naive scalarization and another to perform code transformations to restore the semantics of the program if the initial scalarization was invalid. Using the dependence information produced by the methods described in this paper, we propose a new algorithm that eliminates the need for the first pass and is able to determine a valid scalarization before attempting any transformations.

5.2 One-Pass Scalarization

Our new scalarization algorithm begins by performing dependence analysis directly on the Fortran 90 array statements. When attempting to scalarize an array statement, we only need to be concerned with the scalarization dependences of that statement on itself. As discussed in Section 2.2, such dependences will always be antidependences and may contain any of the three direction specifiers in triplet positions. If we were to perform naive scalarization on a triplet that has a forward ($<$) or loop independent ($=$) antidependence, the resulting loop will have an equivalent antidependence. However, if we were to naively scalarize a triplet that carried a backward ($>$) antidependence, the resulting loop will carry a forward true dependence indicating an incorrect scalarization. Thus we must be careful to address the antidependences that contain an "$>$" in the position corresponding to a triplet we are scalarizing.

Our algorithm will proceed to scalarize the statement one triplet at a time, paying particular attention to those triplets that carry backward antidependences. There are

```
A(2:N-1,2:N-1) = A(1:N-2,3:N)          (>,<)
               + A(3:N,3:N)            (<,<)
               + A(1:N-2,2:N-1)        (>,=)

(a) array statement                (b) dependence vectors
```

Figure 5: One-pass scalarization example.

```
DO J = 2, N-1
  DO I = N-1, 2, -1
    A(I,J) = A(I-1,J+1)
           + A(I+1,J+1)
           + A(I-1,J)
  END DO
END DO
```

Figure 6: Generated scalar code.

several methods we can use to handle these dependences; basically the same methods that the two-pass algorithm uses to address loop carried true dependences.

First we can choose the order in which the triplets will be scalarized. If we choose a triplet position that contains only "<" and "=" elements in the scalarization dependences, we can perform a naive scalarization of that triplet and know that it is correct. Afterward we can eliminate from further consideration those dependences which contained an "<" in that position, since those dependences are carried by the scalarization loop. This is advantageous when the eliminated dependences contained ">" elements in other positions.

Second, if all dependences contain either a ">" or "=" in a given position, the corresponding triplet can be correctly scalarized with a reversed loop. Again, those dependences that were carried at that triplet position can be eliminated. Failing these, we can continue to attempt all the transformations that the two-pass algorithm utilized, including prefetching.

If there are triplets remaining that cannot be scalarized by any of the transformations, we generate a temporary array whose size equals the remaining array section. We then create two adjacent loop nests for the remaining triplets. The first nest performs the desired computation and stores it in the temporary array, and the second copies the results from the temporary array into the destination array.

As an example, consider the statement in Figure 5a and its corresponding scalarization dependences in Figure 5b. After scanning the dependences, we see that the second triplet can safely be scalarized using the naive method. This eliminates the first two dependences from further consideration since they both contain an "<" in the position corresponding to the second triplet. That leaves us with a single dependence of $(>, =)$ and only the first triplet to scalarize. The dependence vector quickly tells us that the remaining triplet can safely be scalarized by generating a reversed loop. The resulting code is shown in Figure 6.

6 Conclusion

In this paper, we have presented a methodology to extend dependence testing to directly analyze data dependences arising from array-section references. The testing procedures presented in this paper were designed to fit smoothly into the framework of an existing dependence testing system; in particular, we have augmented the definitions of complexity and separability to include subscripts containing triplet notation. This extension to dependence analysis will give Fortran 90 and HPF compilers analysis capabilities not previously available, allowing them to make decisions and perform transformations at the Fortran 90 level before scalarizing the program into Fortran 77 code. As a first application, we have shown how this information allows the compiler to perform the scalarization process more efficiently.

Acknowledgments

We'd like to thank Gina Goff and Chau-Wen Tseng for many helpful discussions. This work was supported in part by the IBM Corporation and ARPA contract #DABT63-92-C-0038. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran 90 Handbook*. McGraw-Hill, New York, NY, 1992.

[2] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*. PhD thesis, Dept. of Computer Science, Rice University, April 1983.

[3] 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.

[4] J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

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

[6] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[7] D. Callahan. Dependence testing in PFC: Weak separability. Supercomputer Software Newsletter 2, Dept. of Computer Science, Rice University, August 1986.

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

[9] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[10] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[11] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[12] Z. Li, P. Yew, and C. Zhu. Data dependence analysis on multi-dimensional array references. In *Proceedings of the 1989 ACM International Conference on Supercomputing*, Crete, Greece, June 1989.

[13] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[14] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.

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

[16] M. J. Wolfe and C.-W. Tseng. The Power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

[17] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, New York, NY, 1991.