From the Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), Las Vegas, NV, June 2001

Evaluation of Array Syntax Dependence Analysis

Gerald Roth
Department of Math and Computer Science
Gonzaga University
Spokane, WA, U.S.A.
(*roth@cps.gonzaga.edu*)

Abstract *Dependence analysis and dependence information are critical components of many optimizing and parallelizing compilers. Recently, there has been an increased interest in extending dependence analysis to also cover Fortran 95 array section references. This analysis can then be used in making advanced scalarization, loop fusion, and array contraction decisions. This paper reviews the methods of performing array section dependence analysis, discusses a simplified analysis method, and then evaluates the effectiveness of the simplified method.*

Keywords: dependence analysis, Fortran95, array syntax, scalarization, loop fusion

1 Introduction

In a previous paper of ours, we presented methods for performing dependence analysis on Fortran 95 array syntax [9]. The methods presented in that paper are designed to accurately report data dependence information resulting from any situation where array subscripts contain array syntax notation. In addition, the paper shows how dependence vectors can be extended to hold the additional information, and also how that information can be used to perform advanced compiler transformations.

In this paper we present a simplified array syntax dependence analysis algorithm. The simplified algorithm is very quick and efficient, and it does not require extensive support routines as our previous methods did. However, the simplified algorithm results in some loss of accuracy. We discuss the uses of the data dependence information produced and why the loss of some accuracy has minimal affect on the transformations which use the information. Finally, we discuss the actual implementation of this simplified strategy in two production compilers and report experimental results.

1.1 Fortran 95

Fortran 95 [1] is 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 Fortran 95 has raised the level of abstraction from the strictly scalar constructs of Fortran 77, thus making it 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 optimize by standard analyses and transformations.

It is assumed that the reader is familiar with Fortran 95, especially with the execution semantics of array operations. In Fortran 95, operations deal with their operands as unitary objects, even when they are arrays or array subsections. 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 would 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, that is 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.

2 Dependence Analysis

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, 11, 12].

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 analysis is the process of determining whether a data dependence exists between two statements [4]. Dependence analysis 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.

Recent works [6, 7, 9] show how dependence analysis can be directly performed on array-section references, and how direction or distance vectors can be modified to contain the additional information. In our previous paper, we describe how to extend a partition-based dependence analysis algorithm[5], such as the one used in the analysis and transformation systems at Rice University, to include the analysis of array syntax subscripts. A partition-based analysis scheme depends upon the ability to classify array subscripts based on two orthogonal criteria: *complexity* and *separability*. Complexity refers to the number of distinct loop induction variables that appear within a subscript. Separability refers to whether or not different subscript positions contain common induction variables. Once subscripts are classified, then a hierarchy of dependence tests that are both quick and accurate is chosen for any given situation.

Our previous paper showed how to extend the concepts of complexity and separability to include array section references. From there we presented a battery of tests that could be applied to array-section references depending upon their complexity and separability classifications. These tests depend upon the availability of multiple advanced analysis algorithms already present in the Rice compilation system. The final result is a dependence analysis system for array syntax references that is highly accurate and efficient to some degree.

3 Simplified Strategy

In this section we first present the possible uses of information resulting from the dependence analysis of Fortran 95 array syntax, and exactly what information is required for those applications. Then we present a simplified dependence analysis strategy which satisfies those requirements.

```
X(1:256) = X(1:256) + 1.0
```

(a) array statement

```
DO I=1, 256
  X(I) = X(I) + 1.0
END DO
```

(b) scalarized code

Figure 1: Scalarization example.

```
X(2:255) = X(1:254) + X(2:255)
```

(a) array statement

```
DO I=2, 255
  X(I) = X(I-1) + X(I)
END DO
```

(b) naively scalarized code

Figure 2: Invalid scalarization example.

3.1 Applications of Dependence Information

At some point during the compilation of a Fortran 95 program, array assignment statements must be translated into serial DO-loops. This process is known as *scalarization* [3, 11]. The transformation replaces each array assignment statement with a loop nest containing a single assignment statement in which all array references contain only scalar subscripts. An advanced scalarizer will also attempt to perform loop fusion [8] and array contraction [6] at this time.

For example, consider the array assignment statement shown in Figure 1(a). Scalarization translates the statement into the code shown in Figure 1(b), which iterates over the specified 256 elements of the array `X`.

Unfortunately, the naive translation of array statements into serial loops is not always safe. The Fortran 95 semantics for an array assignment statement specify that all right-hand side array elements are read before any left-hand side array elements are stored. Thus a naive translation of the code shown in Figure 2(a) into the code shown in Figure 2(b) is incorrect, since on the second and subsequent iterations of the `I` loop the reference `X(I-1)` accesses the new values of the array `X` assigned on the previous iteration. This violates the "load-before-store" semantics of the Fortran 95 array assignment statement.

Fortunately, data dependence information can tell us when the scalarized loop is correct. Allen and Kennedy [3] have shown that a scalarized loop is correct if and only if it does not carry a true dependence. Additionally, loop fusion of conformable scalarized loops is always safe when there is no resulting loop carried true dependences. Similarly, array contraction is possible when all dependences are loop-independent.

As can be seen, the dependence analysis of Fortran 95 array syntax can be used to determine if a naive scalarization of an array assignment statement is safe. It can also be used to determine when loops can be fused during the scalarization process and when temporary arrays can be contracted into scalar variables. In all of these cases, it is most important to determine whether a data dependence is loop-carried or loop-independent. It is less important to determine the exact dependence distance or direction when a dependence is loop-carried.

Finally, we note that the scalarization of array assignment statements typically occurs as a final step of the front end of a Fortran 95 compiler as it produces the intermediate representation that is passed to the optimizer or back end of the compiler. Since the advanced analysis routines found in the optimizer only operate on the lower intermediate representation, they typically cannot be used by the scalarizer. Due to this, the scalarizer does not have access to the advanced analysis routines of the optimizer. In particular, any dependence analysis algorithm that operates within the front end of the compiler often does not have access to a symbolic analyzer that can evaluate expressions; such a capability is critical to a

```
X(:,:,NEW) = X(:,:,OLD) + Y(:,:)
```

Figure 3: Ignoring a scalar subscript.

dependence analyzer that operates within the optimizer. It is within this context of an advanced scalarizer in the compiler front end that our simplified strategy is intended to work.

3.2 Simplified Analysis of Array Syntax

Since the goal of our dependence analysis system is to support the scalarization and fusion of array statements, it is not required to disprove all false dependences that may exist between two array references; a conservative estimate is sufficient. A fusing scalarizer, whose goal is to eliminate compiler generated temporary arrays and fuse loops when possible, only really needs to know whether the dependences that are generated due to the array syntax subscripts are loop carried or loop independent. A scalarizer can avoid generating a temporary array and can fuse loops when the dependences are loop independent.

Thus it is conservative and safe to ignore the scalar subscripts that appear in an array reference. Such scalar subscripts can be ignored since we are not necessarily interested in disproving a data dependence, but rather we are interested only in whether an array syntax dependence is loop carried or not.

It is true that if the scalar subscripts are also analyzed that data dependences can be disproved or shown to be carried by outer loops, but we will see in the experimental results that this extra capability is not typically required.

As an example, consider the code shown in Figure 3. If the compiler were able to prove that the scalar variables NEW and OLD contained different values then it could prove that there is no data dependence of the statement on itself due to the two references to the array X. However, even if we assume that they contain the same value, our analysis of the array syntax subscripts can still prove that the dependence is loop-independent and thus not a hindrance to naive scalarization or loop fusion.

Given the operating parameters above, our dependence analyzer has a straight forward testing strategy. Given two array references which are to be tested, we simply walk their subscript lists looking for matching triplets while ignoring scalar subscripts. If the matching triplets are not found in the same subscript positions, the analyzer reports a loop carried dependence in all directions for the given triplet. If, however, the matching triplets are found in the same subscript positions, the analyzer then simply compares the two triplet expressions component for component. If the two triplet expressions are identical, then the dependence cannot be loop carried and the analyzer reports that the dependence is loop independent. If any mismatch of triplet components is found, then the analyzer conservatively reports that the dependence is loop carried.

As is evident in this description of the dependence testing strategy, the analyzer is very fast and efficient. All the work can be done in a single walk of the two array references being tested. There is no need to classify subscripts or perform expensive symbolic analysis. The only apparent drawback of the algorithm is its conservativeness, in that it is possible for it to report a dependence where none exists. However, as stated earlier, the primary goal of this analysis is not to prove data independence, but rather to simply prove that a dependence is not loop carried. As we will see in the next section of this paper, this conservativeness has only a small impact on the success of an advanced scalarizer which uses the dependence information produced.

4 Experimental Results

The simplified dependence analyzer of the previous section has been implemented in two production compilers as part of advanced loop-fusing scalarizers. The two compilers are Sun Microsystem's f90 compiler and Cray's initial

```
t(Ng1:2*Ng1-1,:)  = t(Ng1:1:-1,:)
```

Figure 4: Reported loop-carried dependence.

```
M(K,K:N+1) = M(K,K:N+1) / M(K,K)
```

Figure 5: Actual loop-carried dependence.

f90 compiler for the MTA series (Cray's f90 MTA compiler is still in development at the time of this writing).

To measure the effectiveness of the dependence analyzer, we collected a set of Fortran 95 benchmarks that make heavy use of array syntax statements. The benchmarks collected include three from the NAS Parallel Benchmark suite (EP, SP, and BT)[1], and six benchmarks from the Quetzal Suite (Channel, Gas_Dyn, Monte_Carlo, Scattering, Fatigue, and Capacita). Each test case was compiled with an advanced scalarizer and the execution of the dependence analyzer was traced.

The nine benchmarks contained 557 array statements requiring scalarization and for which fusion was attempted. Within these statements, a total of 658 data dependences were reported by the simplified analyzer. Of these dependences, 645 were found to be loop independent, and thus were not a hindrance to naive scalarization or fusion of loops. Of the thirteen dependences that were reported as being loop carried, eleven were inter-statement dependences and thus they did not require a compiler temporary to resolve; instead they only prevented the fusion of adjacent loops. The final two intra-statement, loop carried dependences resulted in array temporaries and copy loops being generated for only two of the 557 array statements.

Both loop carried dependences were reported in the same test case and both were a result of assignment statements such as the one in Figure 4. As can be seen, one triplet is counting forward and the other is counting backward. Actually, this statement has a loop independent dependence on the first triplet value and then no dependences after that. Thus the simplified strategy was too conservative in this case, but without the power of a symbolic analyzer there is not much else that can be done for this statement.

We also tested our advanced scalarizer and simplified dependence analyzer on the NAG Fortran 95 test suite, which contains approximately 530 test cases. This test suite is intended to test the full functionality of a Fortran 95 compiler, and thus was not as interesting for the scalarizer as the benchmark test cases were. Within the NAG suite, the scalarizer encountered 791 array statements requiring scalarization. Within all those array statements, only 25 data dependences were reported. Of the 25 dependences, 20 were found to be loop independent by the simplified strategy and only five were reported as loop carried.

An example of a loop carried dependence from the NAG suite is shown in Figure 5. As can be seen, there is an actual loop carried dependence on $M(K, K)$ which prevents a naive scalarization of this statement.

These results show that the simplified dependence analysis strategy was very successful in determining the loop independence of the dependences encountered. As was stated earlier, determining loop independence, rather than disproving a dependence, is sufficient for a scalarizer to perform naive scalarization, loop fusion, and array contraction.

5 Related Work

The ZPL compiler [6], in a manner similar to the work presented here, performs array-level dependence analysis to determine a valid strategy for scalarization and fusion prior to making any code transformations. It also aggressively rearranges array statements to promote loop fusion for the purpose of array contrac-

[1] We used the HPF version of the NAS Parallel Benchmarks, as produced by The Portland Group.

tion. It is closely related to our previous work on compiling HPF [10]. Other than that, there is little published information on performing dependence analysis directly on Fortran 95 array syntax expressions.

6 Conclusion

In this paper we have presented a strategy for performing simplified dependence analysis on array syntax references. The goal of the analysis is to support the scalarization and fusion of array statements by classifying dependences as loop independent or loop carried. The strategy has been implemented in two commercial compilers, and results show that it is highly successful in accomplishing it goals.

Acknowledgments

I'd like to thank Robert Corbett, Larry Meadows, and Prakash Narayan of Sun Microsystems and Gail Alverson, David Callahan, and Mark Niehaus of Cray Inc for their support of this work.

References

[1] J. Adams, W. Brainerd, J. Martin, and J. Wagener. *Fortran 95 Handbook.* The MIT Press, Cambridge, MA, 1997.

[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. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.

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

[5] 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.

[6] E Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[7] G. Roth. *Optimizing Fortran90D/HPF for Distributed-Memory Computers.* PhD thesis, Dept. of Computer Science, Rice University, April 1997.

[8] G. Roth. Advanced scalarization of array syntax. In *Proceedings of the 9th International Compiler Construction Conference (CC'2000)*, Berlin, Germany, March 2000.

[9] G. Roth and K. Kennedy. Dependence analysis of Fortran90 array syntax. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, Sunnyvale, CA, August 1996.

[10] G. Roth and K. Kennedy. Loop fusion in High Performance Fortran. In *Proceedings of the 1998 ACM International Conference on Supercomputing*, Melbourne, Australia, July 1998.

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

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