From the Proceedings of the 9th International Compiler Construction Conference (CC'2000), Berlin, Germany, March 2000

Advanced Scalarization of Array Syntax

Gerald Roth

Dept of Math and Computer Science, Gonzaga University
Spokane, WA 99258
roth@cps.gonzaga.edu

Abstract. One task of all Fortran 90 compilers is to scalarize the array syntax statements of a program into equivalent sequential code. Most compilers require multiple passes over the program source to ensure correctness of this translation, since their analysis algorithms only work on the scalarized form. These same compilers then make additional subsequent passes to perform loop optimizations such as loop fusion. In this paper we discuss a strategy that is capable of making advanced scalarization and fusion decisions at the array level. We present an analysis strategy that supports our advanced scalarizer, and we describe the benefits of this methodology compared to the standard practice. Experimental results show that our strategy can significantly improve the runtime performance of compiled code, while at the same time improving the performance of the compiler itself.

1 Introduction

Fortran 90 and High-Performance Fortran (HPF)[7] 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 analysis 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 then show how this dependence information can be used to perform advanced scalarization and loop fusion transformations. We conclude with a brief look at some preliminary performance results obtained by incorporating these methods into Sun Mircosystem's f90 compiler.

```
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

Fig. 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

Fig. 2. Invalid scalarization example.

2 Standard Scalarization and Optimization

At some point during the compilation of a Fortran 90 program, array assignment statements must be translated into serial DO-loops. This process is known as *scalarization* [2, 11, 15]. 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.

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 90 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 90 array assignment statement.

Fortunately, data dependence information can tell us when the scalarized loop is correct. Allen and Kennedy [2] 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 a scalar loop nest.
2. Compute the data dependences of the resulting code.
3. While a scalarized 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 and copy loops. In the example above, loop reversal can be used to change the loop-carried true dependence into a loop-carried antidependence, thus creating a valid scalarization. The interested reader is referred to Allen and Kennedy [2] for a complete discussion. It is noted that the Allen and Kennedy algorithm requires two passes over the code: one pass to perform the naive scalarization followed by dependence analysis and another pass to perform code transformations to restore the semantics of the program if the initial scalarization is invalid.

After scalarization, a program will consist of many loop nests, each containing a single assignment statement. If the goal of a Fortran 90 compiler is to produce code for array expressions that is competitive with code produce for scalar programs by Fortran 77 compilers, it is critical that the Fortran 90 compiler do a good job of fusing these loops when possible. *Loop fusion* [3, 16] not only reduces the total amount of loop overhead, but more importantly it can significantly increase the possibility of data reuse in a program. The importance of loop fusion in the compilation of Fortran 90 array statements cannot be over emphasized [9].

In previous work on loop fusion [5, 14, 16] two adjacent loops are candidates for fusion if their headers are *conformable* and no *fusion-preventing dependences* exist. Two loop headers are conformable if they specify the same number of iterations. A data dependence between two loops is fusion-preventing if after fusion the dependence becomes loop-carried and its direction is reversed [14].

By using loop fusion, in conjunction with other transformations such as statement substitution and array contraction, it is possible for a Fortran 90 compiler to generate code for a block of array assignments that is equivalent to the code produced by a Fortran 77 compiler for a corresponding hand-coded loop nest.

3 Advanced Scalarization and Loop Fusion

In this section we will describe an advanced scalarizer that performs dependence analysis at the array syntax level, directly scalarizes array statements in a single pass, and is capable of directly generating fused loops during the scalarization process.

3.1 Array Section Dependence Analysis

Before beginning a discussion on dependence analysis, it is important to clarify some terminology that is used in this section. An *array reference* is a subscripted variable reference. A *subscript* is one element from a subscript list. A triplet, as seen in Figure 1(a), is one type of subscript. It is assumed that Fortran 90 whole array references, array references without a subscript list, are represented internally within the compiler to include a subscript list containing the appropriate number of null triplets.

```
    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
```

Fig. 3. Fortran 90 code fragment.

Data dependences are often represented using *direction vectors* and/or *distance vectors* [15]. The direction vector is an ordering vector, containing $<$, $=$, $>$, or $*$, that specifies the relation of the source and target iterations involved in the dependence. The distance vector contains the vector difference of the source and target iterations. 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 work we discuss only direction vectors, although the algorithms discussed could easily be adapted to work with distance vectors.

Direction vectors are useful in determining if a dependence is *loop-carried* or *loop-independent* [1]. 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.

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. In most cases these vector elements are only considered when the two array references are conformable, in which case they have the same number of triplets. These new direction vector elements appear to the right of those elements corresponding to surrounding loops. We 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 3. 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. This fragment of code contains the following four dependences: $S_1\overline{\delta}_{(=,>,=)}S_1$, $S_1\overline{\delta}_{(=,=,=)}S_1$, $S_1\delta_{(=,>,=)}S_2$, and $S_2\overline{\delta}_{(<,=,=)}S_1$, where δ indicates a true (or flow) dependence, and $\overline{\delta}$ indicates an antidependence.

3.2 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* [11]. 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 by others [2, 4].

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 is always an antidependence (ignoring input dependences). Secondly, all inter-statement scalarization dependences flow in lexicographical order, from the earlier statement to the later statement. 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.

However, this does not mean that we can ignore scalarization dependences. These dependences play an important role when the compiler scalarizes the Fortran 90 program into its Fortran 77 equivalent. This aspect of the dependences is addressed in more detail in the next subsection.

Consider again the code in Figure 3. This fragment of code contains three 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.

The actual dependence testing algorithms used to test array syntax references are described in previous works [10, 11] and are not presented here.

3.3 One-Pass Scalarization

As described in Section 2, the typical scalarizer requires two passes over the code to produce a valid scalarization: one to perform a naive scalarization and a second to apply code transformations to restore program semantics when necessary. Using the dependence information described in the preceding subsection, 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.

Our new scalarization algorithm begins by performing dependence analysis directly on 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 the preceding subsection, such dependences are always antidependences and may contain any of the three direction

specifiers in triplet positions. If we perform naive scalarization on a triplet that has a forward (<) or loop independent (=) antidependence, the resulting loop has an equivalent dependence. However, if we naively scalarize a triplet that carries a backward (>) antidependence, the resulting loop carries 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 proceeds to scalarize the statement one triplet at a time, paying particular attention to those triplets that carry backward antidependences. There are 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 are 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 scalarized loop. This is advantageous when the eliminated dependences contained non-"=" 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 4(a) and its corresponding scalarization dependences in Figure 4(b). 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 direction vector quickly tells us that the remaining triplet can safely be scalarized by generating a reversed loop. The resulting code is shown in Figure 5.

3.4 Scalarization Plus Loop Fusion

As discussed in Section 2, loop fusion is an important optimization associated with scalarization. After scalarization, a program will consist of many loop nests, each containing a single assignment statement. The duty of loop fusion is to combine these loop nests when possible to reduce loop overhead and to increase

```
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
```

Fig. 4. 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
```

Fig. 5. Generated scalar code.

the possibility of data reuse. As prevously discussed, loop fusion is safe when the loop headers are conformable and no fusion-preventing dependences exist.

These criteria can be non-trivial to verify in the general case of fusing Fortran 77 loops; although the task can be easier for scalarized loops since the structure of the loop headers is completely under the control of the compiler. Unfortunately, in many Fortran 90 compilers, scalarization and loop fusion are far removed from one another, the former occuring in the compiler front end while the latter occurs in the compiler back end. This seperation has several drawbacks. First, each must compute the necessary data dependence information anew, since it is not likely to remain valid given the transformations are so distant from each other. Secondly, intermediate transformations that alter the loops may impede fusing the scalarized loops.

To address this problem, we have combined scalarization and fusion into a single integrated optimization. The goal of this unified optimization is to scalarize multiple array statements into a single loop nest, while maintaining program semantics and avoiding the generation of array temporaries at all costs [1].

The fusing scalarizer depends upon scalarization dependences, just as our single-statement scalarizer did. However, we are now considering inter-statement dependences along with the intra-statement dependences. As with scalarizing a single statement, we must pay special attention to triplets that carry backward (>) dependences, whether they are true dependences or antidependences. If two statements that have an inter-statement, backward-carried dependence are naively scalarized and fused into a single loop nest then the dependence becomes

[1] The cost of allocating and freeing array temporaries far outweighes any possible benefits of loop fusion, and thus temp arrays should be avoided whenever possible.

forward-carried in the opposite direction and its meaning becomes inverted (*e.g.*, true becomes anti and vice versa).

As an example, consider again the code in Figure 3. This fragment of code contains the inter-statement scalarization dependence $S_1\delta_{(=,>,=)}S_2$. If the two statements were naively scalarized and fused, the dependence relationship of the resulting code would be $S_2\overline{\delta}_{(=,<,=)}S_1$, indicating that the transformation did not maintain program semantics. However, just as in scalarizing a single statement that had backward dependences, it is possible to use loop reversal to maintain the dependence relationship and program semantics.

Our fusing scalarizer has the following general outline. First, all array statements requiring scalarization are identified. Each array statement is then considered in lexicographical order. Dependence analysis is performed on the statement to identify all intra-statement scalarization dependences. If the dependences indicate that an array temporary is required for valid scalarization, the statement is scalarized by itself (with an array temp) and we move onto the next statement.

However, if the statement can be scalarized without an array temporary, we begin considering fusing subsequent array statements. This is accomplished by seeding a *fusion group* with the given statement. The purpose of the fusion group is to identify those array statements that can be scalarized and fused as a single group. Subsequent array statements are then considered for inclusion in the fusion group. To be included, an array statement must satisfy the following requirements:

1. The array statement must be adjacent to the last statement in the fusion group.
2. The statement must be conformable with the other statements in the group. Since all statements already in the group are known to be conformable, it is only necessary to test the candidate statement against any one representative of the group. Note that due to the typically regular nature of array syntax used in practice, this criteria is usually quite easy to verify, unlike determining conformance of general Fortran 77 loop headers.
3. The candidate statement must be scalarizable without a temporary array. This is determined by performing dependence analysis on the candidate statement and examining all intra-statement scalarization dependences as described in the preceding subsection.
4. Finally, the entire fusion group must be scalarizable into a single loop nest without a temporary array if the candidate is included. This can be determined by performing dependence analysis between the candidate statement and each statement in the fusion group, adding all inter-statement dependences to the set of dependences already present in the fusion group. The intra-statement dependences identified in the preceding step are also included. Then the entire set of dependences is examined to determine if a valid scalarization exists.

Statements are added to the fusion group in this manner, one at a time, until a statement is encountered which violates one of the above criteria. At

```
S1:  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)
S2:  B(2:N-1,2:N-1) = A(2:N-1,2:N-1) + A(2:N-1,1:N-2) + B(1:N-2,2:N-1)
```

Fig. 6. One-pass scalarization plus fusion 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)
    B(I,J) = A(I,J) + A(I,J-1) + B(I-1,J)
  END DO
END DO
```

Fig. 7. Generated scalar code.

that point, the entire fusion group is scalarized into a single loop nest, with the scalarization dependences dictating the order and format of the loops. The strategy is the same as that used in scalarizing a single statement as described in the preceding subsection. After the fusion group has been scalarized the process begins again, starting with the last statement that was not added to the fusion group.

As an example, consider the two statements in Figure 6. Statement S_1 has the following set of intra-statement scalarization dependences: $S_1\overline{\delta}_{(>,<)}S_1$, $S_1\overline{\delta}_{(<,<)}S_1$, and $S_1\overline{\delta}_{(>,=)}S_1$. Statement S_2 has this intra-statement scalarization dependences: $S_2\overline{\delta}_{(>,=)}S_2$. And finally, the inter-statement scalarization dependences are: $S_1\delta_{(=,=)}S_2$, and $S_1\delta_{(=,<)}S_2$.

As was shown in the preceding subsection, our algorithm easily determines the S_1 can be scalarized without an array temporary. It would thus start a fusion group. Next S_2 is considered. Since S_2 has only one intra-statement dependence, we know we can scalarize it without the aid of a temporary array. We must next consider all the scalarization dependences between S_1 and S_2, both inter-statement and intra-statement dependences. These dependences indicate that the second triplet can directly be scalarized with a naive loop. After eliminating all dependences that are carried by that naive loop, the remaining dependences indicate that the final triplet can be safely scalarized by generating a reverse loop. Since scalarization is possible without an array temporary, statement S_2 will be added to the fusion group, and the scalarization process continues. If there were no more statements to be considered, code for the fusion group containing S_1 and S_2 would be generated as seen in Figure 7.

4 Experimental Results

The algorithms and scalarization/fusion strategy presented in this paper have been partially implemented in Sun Microsystem's f90 compiler. Previously, the

compiler had a very simple scalarizer that always generated a compiler temporary array and multiple loop nests per array statement. The compiler still uses the simple scalarizer at lower optimization levels.

To measure the performance improvements attained with the new advanced scalarizer, we collected a set of Fortran 90 benchmarks that make heavy use of array syntax statements. The benchmarks collected include three from the NAS Parallel Benchmark suite (EP, SP, and BT)[2], and six benchmarks from the Quetzal Suite (Channel, Gas_Dyn, Monte_Carlo, Scattering, Fatigue, and Capacita). Each test case was compiled twice, once with the existing simple scalarizer and once with the new advanced scalarizer. All test cases were compiled at `-O4`, regardless of which scalarizer was used, so that the compiler's optimizing backend would attempt a full compliment of scalar optimizations, including loop fusion, after scalarization was performed.

The nine benchmarks contained 557 array statements requiring scalarization. The simple scalarizer generated an array temporary and multiple loop nests for each of these array statement. In contrast, the advanced scalarizer was able to scalarize all but two without the need for an array temporary and multiple loops. In addition, the loop fusion capability of the advanced scalarizer was able to fuse 326 of the remaining 555 array statements into 83 separate loop nests; that is a reduction of 44% in the total number of loop nests generated. The average fused loop contained 4 array statements, while the maximum number of statements fused into a single loop nest was 8.

The run-time performance results are shown in Figure 8. As can be seen, the speed-ups attained on the nine benchmarks ranged from 1.2 all the way up to a factor of 21.8 for the Capacita benchmark [3]. The average speed-up, disregarding the Capacita number, was 2.3. The following are the primary reasons for the significant improvements in runtime performance:

1. The advanced scalarizer generated an array temporary for only two of the 557 array statements scalarized. Whereas the simple scalarizer generated an array temporary and multiple loop nests for each statement.
2. The advanced scalarizer was successful in fusing multiple array statements into single loop nests while the nests were being generated. Due to the presence of the array temporaries, the optimizing back end was not able to fuse many loops produced by the simple scalarizer.
3. The advanced scalarizer generated in-line code for several array reduction intrinsics (*e.g.*, SUM/PRODUCT, MINVAL/MAXVAL, and ANY/ALL), as well as the DOT_PRODUCT intrinsic function. The simple scalarizer depended upon library routines to handle all such intrinsics.
4. The advanced sclarizer produced code that looks exactly like the code the compiler front end would produce for equivalent Fortran 77 code. This assists

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

[3] The dramatic improvement for the Capacita benchmark is the result of extremely poor code generated by the simple scalarizer for a single statement that combined an array reduction intrinsic with a whole array operation.

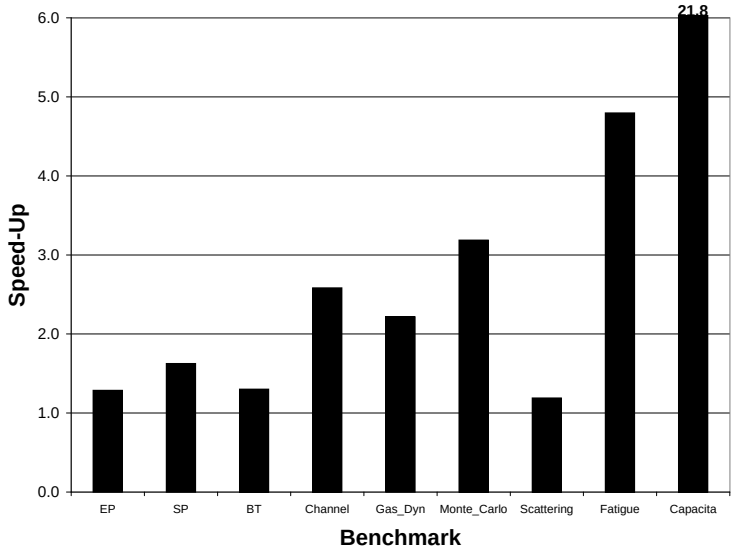

Fig. 8. Run time performance speed-ups

the backend since it is geared toward optimizing standard Fortran 77 loop nests.

We now consider the affects of the advanced scalarizer on compile time. One might expect that the analysis required to perform advanced scalarization and fusion to come at a high cost. However, just the opposite is true. Performing dependence analysis at the array statement level is generally inexpensive, since the array syntax used in practice tends to be very regular in nature. And since the fusing scalarizer produces fewer loop nests, there is less work to be done by all subsequent compiler phases.

For all nine benchmarks discussed previously, the compiler actually ran faster with the advanced scalarizer than without it. Figure 9 summarizes the compile time speed ups obtained when the advanced scalarizer was used. Speed-ups ranged from 1.2 to 7.8; the average was 3.2.

5 Related Work

Many people have recognized the need to fuse loops generated by the scalarization of Fortran 90 array statements [2, 6, 13, 16]. However, all of these only consider fusion *after* array statements have been scalarized.

The ZPL compiler [8] is an exception. In a manner similar to the work presented here, the ZPL compiler performs array-level analysis to determine a valid strategy for scalarization and fusion prior to making any code transformations.

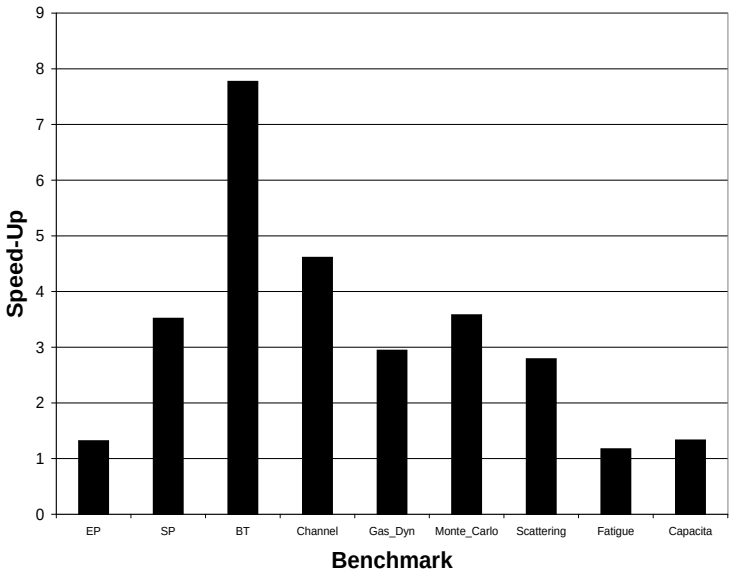

Fig. 9. Compile time performance speed-ups

It aggressively rearranges array statements to promote loop fusion for the purpose of array contraction. It is closely related to our previous work on compiling HPF [12]. However, in both of these cases the target architecture was a distributed-memory parallel machine. For that reason, neither of these prior works would perform any fusion if there were inter-statement loop carried dependences, since such dependences would entail the addition of interprocessor communication into the scalarized loop nest, resulting in slower performance. For example, neither compiler would have performed the fusion displayed in Figure 7.

6 Conclusion

In this paper we have presented a strategy for combining scalarization and loop fusion into a single integrated algorithm. The strategy relies upon advanced analysis and transformations performed at the level of Fortran 90 array statements. The strategy has been implemented in a commercial compiler, and preliminary performance results show a significant improvement in both compile time and run time over the naive strategy previously implemented.

Future work on this scalarization strategy would include the analysis to permit statement reordering so as to promote additional loop fusion, as well as applying this strategy to the scalarization and fusion of array statements in WHERE-blocks.

Acknowlegments

I'd like to thank Robert Corbett, Larry Meadows, and Prakash Narayan of Sun Microsystems for their support of this work.

References

1. 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.
2. J. R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):1290–1317, October 1992.
3. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
4. 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.
5. D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Dept. of Computer Science, Rice University, March 1987.
6. M. Chen and J. Cowie. Prototyping Fortran-90 compilers for massively parallel machines. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.
7. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.
8. 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.
9. John D. McCalpin. A case study of some issues in the optimization of Fortran 90 array notation. *Scientific Programming*, 5:219–237, 1996.
10. G. Roth. *Optimizing Fortran90D/HPF for Distributed-Memory Computers.* PhD thesis, Dept. of Computer Science, Rice University, April 1997.
11. 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.
12. 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.
13. G. Sabot, (with D. Gingold, and J. Marantz). CM Fortran optimization notes: Slicewise model. Technical Report TMC-184, Thinking Machines Corporation, March 1991.
14. J. Warren. A hierachical basis for reordering transformations. In *Conference Record of the Eleventh Annual ACM Symposium on the Principles of Programming Languages*, Salt Lake City, UT, January 1984.
15. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.
16. M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.