Scalarizing Fortran 90 Array Syntax

Yuan Zhao   Ken Kennedy

TR01-373, Technical Report
Computer Science Department, Rice University

March 30, 2001

1 Introduction

Array syntax is an important feature introduced in Fortran 90 [1, 4]. It adds more expressive power to the language by allowing operations and assignments on the array sections. Programmers will benefit from this new feature directly by writing simple and concise programs. The remaining work is left to compilers that finally compile these statements with array syntax onto targeted machines. This compiling process is usually called scalarization.

One problem facing Fortran 90 compilers is the temporary arrays generated during the scalarization in order to maintain the semantics of the array syntax [10]. For many years, techniques have been developed to avoid the use of temporary arrays in certain cases. In this paper, we will present two new approaches to improve the scalarization. Our approaches show that in certain cases, the temporary array can be eliminated or reduced, when exact dependence information can be obtained. By doing so, our approaches help improve the memory hierarchy performance on a single scalar machine, in terms of the memory bandwidth.

In what follows, we briefly explain array syntax, data dependence and scalarization in this section. Section 2 gives some previous work. We present our approaches in Section 3 and Section 4, followed by the performance study in Section 5.

1.1 Array Syntax

Array syntax refers to the use of subscript triplets to represent an array or a subsection of an array. The triplet has the syntax form: [expression]:[expression][:expression], where from left to right, expressions represent lower bound, upper bound and stride, respectively. If a lower bound or an upper bound are omitted, they refer to the lower bound or the upper bound of the array at the declaration. The default stride is 1. For example, A(3:4) represents elements A(3) and A(4), while A(1:5:2) refers to array elements A(1), A(3) and A(5). These collections of array elements represented in array syntax are also called array sections.

Array syntax can be used in array assignments, where statements, where constructs, and intrinsic functions. In this paper, we focus on the array assignments where temporary arrays could be allocated during scalarization. The left hand side of array assignments are array sections. An valid array assignment requires that the result of right hand side be the same shape as the left hand side or the result of right hand side be a scalar. Here are two examples, `A(1:4) = B(2:5) + C(2:5)` adds elements of B and C from 2 to 5 and store the results to A at location from 1 to 4, `A(:) = 0` initializes all elements in A to 0.

1.2 Scalarization and Data Dependence

When compiling array assignments to scalar machines, which can only handle scalar operations, operations of the array assignments can only be done on one element at a time. Thus array assignments have to be translated into loop nests. This process is called scalarization.

The simplest idea of scalarization is to iterate through all triplets in the subscript for the loop, and to use the corresponding loop index at the triplet location. For the first example we used in section 1.1, the transformed code looks like: `DO I = 1, 4 ; A(I) = B(I+1) + C(I+1) ; ENDDO` We call this method naive scalarization.

Scalarization must preserve the semantic of array assignments. Similar to the vector instructions on vector machines, the semantic of array assignments is "fetch before store", *i.e.* all elements on the right hand side are fetched first, followed by the computation, followed by storing the result into the left hand side. Under this semantic, the naive scalarization method may be wrong. For example, `A(2:5) = A(1:4) + A(3:6)` has different meaning from `DO I = 1, 4; A(I+1) = A(I) + A(I+2); ENDDO`.

Allen and Kennedy [3] pointed out that the scalarized code is correct only if the loop nest does not carry any true dependence. There is a dependence between two statements if two statements access the same memory location and at least one of them stores into it, and there is an feasible execution path from one statement to the other [2]. It is said to be true dependence if one statement read the result written by the other statement. The dependence is said to be loop carried if two accesses of the same memory location occur at different loop iterations. In our last example `DO I = 1, 4; A(I+1) = A(I) + A(I+2); ENDDO`, there is a loop carried true dependence because the statement stores the result which is read by the same statement one iteration later.

When there are multiple dependences in a loop nest, we usually represent them in a dependence matrix, where each row is a dependence vector and each column corresponds to a loop, starting from the outmost one. A dependence is carried by the first non-"=" entry (column, loop). In this paper, all algorithms are based on the scalarization dependence matrix, which only contains loop carried true and loop carried anti dependences. Note that for an array assignment statement, there will be no output dependence carried by the scalarized loop nest. Since there are only true and anti dependences in our dependence matrix, we omit the type associated with each dependence vector in ordinary dependence matrix, instead we use "$<$" for true dependence, "$>$" for anti dependence, if they appear in the first non-"=" column. We also use "=" for dependence distance of 0, "$*$" for dependence direction that either cannot be determined at compiler time or could be any direction. For a scalarization to be correct, the first non-"=" entries of all dependence vectors in the scalarization dependence matrix of loop nest should be "$>$", which means, the scalarized loop nest only carries anti dependences, if there are any. We only consider the dependences caused by the original array only, not including the ones introduced by the temporary arrays the algorithms may create.

2 Previous Work

In this section, we present existing techniques to ensure the correctness of scalarization and how they try to improve the scalarized code in terms of memory hierarchy performance.

2.1 Full Size Temporary Array

Allocating a full size temporary array is always a failsafe method to guarantee the correctness of the scalarized code [10]. For this method, an array assignment is splitted into two loop nests. In the first loop, all right hand side operands are fetched and computed, and the corresponding results are stored into the temporary array. In the second loop, all results are stored back into the left hand side operands from the temporary. This method completely eliminates the true dependence carried by scalarized loops. For the previous example, `A(2:5) = A(1:4) + A(3:6)`, a correct scalarization is:

```
ALLOCATE T(4)
DO I = 1, 4
 T(I) = A(I) + A(I+2)
ENDDO
DO I = 1, 4
 A(I+1) = T(I)
ENDDO
DEALLOCATE T
```

failsafe version with temporary array

Notice that the penalty of this failsafe method is the introduction of a temporary array. This increases the memory bandwidth requirement in terms of the memory hierarchy performance. An intuitive question to ask is that do we really need this temporary to guarantee the correctness. In next several sections, we will discuss existing methods that avoid using the temporary array or at least use one of smaller size.

2.2 Loop Reversal

The idea of loop reversal is that if the loop is run backwards, all true dependences carried by that loop will become anti dependences [9]. Here are two versions of scalarization for array assignment `A(2:11) = A(1:10) + C`, one is a naive version which violates the semantics of the array assignment, while the other is a loop reversal version which preserves the semantics:

```
DO I = 1, 10
 A(I+1) = A(I) + C
ENDDO
```

(a) naive version, wrong

```
DO I = 10, 1, -1
 A(I+1) = A(I) + C
ENDDO
```

(b) loop reversal version, correct

However, loop reversal will turn the loop carried anti dependence into true dependence. Thus, loop reversal technique cannot handle the situation that both true and anti dependences are carried by the same loop.

2.3 Loop Interchange

The array assignment semantics do not require which dimension of the array section should be computed first. Any computation ordering will be fine as long as the "fetch-before-store" is properly maintained. It is possible that for a loop carried true dependence, if we change the loop ordering, it will become an anti dependence carried by another loop. In this loop interchange [2] process, if we turn all loop carried true dependences into loop carried anti dependences and do not introduce any new loop carried true dependence, we will obtain a correct scalarization loop nest without temporary arrays. Here is an example for array assignment, `A(1:10, 2:11) = A(1:10, 3:12) + A(2:11, 1:10)`, we have a naively scalarized version and a version with loop interchange:

```
DO J = 1, 10
 DO I = 1, 10
  A(I, J+1) = A(I, J+2) + A(I+1, J)
 ENDDO
ENDDO
```
(a) naive version, wrong

```
DO I = 1, 10
 DO J = 1, 10
  A(I, J+1) = A(I, J+2) + A(I+1, J)
 ENDDO
ENDDO
```
(b) loop interchange version, correct

In the naive version, `J` loop carries both a true dependence and an anti one. Applying loop reversal won't help. However, if we move `I` loop to the outmost position, we will have two loop carried anti dependences, one by `I` loop, the other by `J` loop. This is a correct version of scalarized code without temporary arrays.

2.4 Input Prefetching

Input prefetching [3] is another technique to get rid of loop carried true dependence. There is a loop carried true dependence because array elements are stored into before we can retrieve the old value out. Therefore, if we can prefetch the value several iterations earlier and put it somewhere else for later use, we can have a correct scalarization. If the dependence distance is d, then the prefetch should be done d iterations earlier. Let the current loop size be N, the total size of all remaining loops including this one be S, we need to allocate a temporary array of size $(d+1)*S/N$. Usually, this technique is only applied when d is a small constant. Therefore, we reduce the size of temporary array by a factor of $N/(d+1)$.

For one of the previous examples, `A(2:5) = A(1:4) + A(3:6)`, there is only one loop carried true dependence, whose distance is 1, we can apply input prefetching to obtain a correct scalarization.

```
DO I = 1, 4
 A(I+1) = A(I) + A(I+2)
ENDDO
```
(a) naive version, wrong

```
ALLOCATE T(2)
T(1) = A(1)
DO I = 1, 4
 T(2) = T(1) + A(I+2)
 T(1) = A(I+1)
 A(I+1) = T(2)
ENDDO
DEALLOCATE T
```
(b) input prefetch version, correct

2.5 A Scalarization Algorithm

Putting above techniques together, we have a complete scalarization algorithm [3]. Notice that we will try to allocate temporary array as innermost as possible since that will give a smaller temporary array with fewer dimensions.

1. Naively scalarize the array assignment into a loop nest
2. Compute the scalarization dependence matrix from the loop nest
3. for i = 1 to n loops in the loop nest
   (a) If i carries no true dependences, continue
   (b) If i carries only true dependences but no anti dependences, apply loop reversal, continue
   (c) Search for candidate loop j in loop nest (i+1, n) such that j carries either no true dependences or only true dependences, interchange loop i and j, apply loop reversal if necessary, continue
   (d) If i only carries only one true dependence with known distance and one or more anti dependences, prefetch input, continue
   (e) Otherwise, we need to allocate an temporary array of the size of remaining loops, split the remaining loops which is the failsafe option, break

3 Loop Alignment

All the previous work we have shown transform the array assignment into one or two loop nests so that none of these loop nests carries any true dependence. The transformations are chosen based on the analysis of the scalarization dependence matrix which is derived from the naively scalarized loop. In this section, we present a new transformation which can be applied during scalarization. This new transformation, loop alignment [3], shares the same goal as other techniques that improves the memory hierarchy performance by saving memory bandwidth.

In what follows, we will describe how the loop alignment works in the context of scalarization, when it can be used and its profitability, and how it compares to other techniques . At the end, an improved version of scalarization algorithm will be given.

3.1 Loop alignment in scalarization

Now we show how loop alignment can be applied during scalarization through an example: `A(2:11) = A(1:10) + A(3:12)`. The following are two versions of loop nests, one is the naively scalarized version which is wrong because the `I` loop carries a true dependence, the other is the failsafe version.

```
DO I = 1, 10
 A(I+1) = A(I) + A(I+2)
ENDDO
```
(a) naive version, wrong

```
ALLOCATE T(10)
DO I = 1, 10
 T(I) = A(I) + A(I+2)
ENDDO
DO I = 1, 10
 A(I+1) = T(I)
ENDDO
DEALLOCATE T
```
(b) failsafe version, correct

From the failsafe version, we observe that two loops cannot be fused together because there is a loop carried backward true dependence, which is a fusion preventing dependence. In order to fuse them, we need to get rid of this dependence. Another observation is that the distance of the dependence is 1. Therefore, if the two loops are aligned by this distance, the loop carried backward true dependence will be removed and the two loops can be fused together. This process is shown as follows:

```
ALLOCATE T(10)
T(1) = A(1) + A(3)
DO I = 1, 9
 T(I+1) = A(I+1) + A(I+3)
ENDDO
DO I = 1, 9
 A(I+1) = T(I)
ENDDO
A(11) = T(10)
DEALLOCATE T
```
(c) peeled (aligned)

```
ALLOCATE T(10)
T(1) = A(1) + A(3)
DO I = 1, 9
 T(I+1) = A(I+1) + A(I+3)
 A(I+1) = T(I)
ENDDO
A(11) = T(10)
DEALLOCATE T
```
(d) fused

After the two loops are fused, we see that we don't really need a temporary array of size 10, 2 will be enough as shown in the following code.

```
ALLOCATE T(2)
T(1) = A(1) + A(3)
DO I = 1, 9
 T(2) = A(I+1) + A(I+3)      computation section
 A(I+1) = T(1)               store section
 T(1) = T(2)                 alignment section
ENDDO
A(11) = T(1)
DEALLOCATE T
```
(e) temporary size reduced

Though we show the whole process in 4 steps, we don't really need to separate the loop alignment, fusion and the reduction of the temporary array size, simply because we can determine the align distance and the temporary array size at the very beginning. For convenience, we merge them into a single routine and call it **loop alignment**. Notice that the final code shape consists of three parts: computation section, store section and alignment section, as shown in the above example.

3.2 Applicability and profitability of loop alignment

Loop alignment can help eliminate loop carried true dependence in the naive version. Since loop reversal doesn't introduce any temporary arrays when a loop only carries true dependence, loop alignment should be applied when a loop in the naive version carries both true and anti dependences. For alignment to be possible, the distances of these true dependences has to be known at compile time. In practice, we only align the dependences with distances of small constants.

Similar to all previous transformations mentioned in Section 2, whether to perform loop alignment is determined by the dependence matrix which consists of the dependences carried by remaining loops. First, we look at whether a loop is alignable. Notice that we only want to align a loop which carries both true and anti dependences. For the entries of the corresponding column in the dependence matrix, the loop is alignable if the following three conditions hold:

1. There are one or more “<”s and one or more “>”s, zero or more “=”s, “>=”s, but no other direction symbols.
2. For all “<”s, their dependence distances are known at compile time and the maximum is less than or equal to a small constant. This small constant is specified by the compiler implementation. We use 3 for our experiments. The maximum is the alignment distance.

However, for the remaining loops, there could be more than one candidate for loop alignment. For a candidate, let the alignment distance be d, its loop size be N and the size of all remaining loops including this one be S, we will have to allocate a temporary array of size $(d+1) * S/N$. Therefore, we choose a candidate that can minimize this expression. Of course, this is not always possible in practice since N could be symbolic and unknown at compile time. In that case, we can just pick the candidate with a smaller d. If the chosen candidate is not in the outmost position for currently remaining loops, interchange it with the current outmost loop. Of course, the columns in the dependence matrix should be exchanged also.

Second, we show how to update the dependence matrix after loop alignment. For every entry in the column corresponding to the aligned loop,

1. If the direction is “>”, keep the entry unchanged.
2. If the direction is “>=” or “=”, change the entry to “>”.
3. If the direction is “<”, change the entry to either “=” or “>”. If the dependence distance is equal to the alignment distance, change the entry to “=”, otherwise, the dependence distance is less than the alignment distance, change the entry to “>”.

After updating entries, remove all dependences that have a “>” in this column from the dependence matrix since these dependences are now all anti dependences carried by the chosen aligned loop. When we move to the next (inner) loop in the loop nest, these dependences need not be considered.

Last, we give the algorithm for loop alignment in scalarization.

1. j = 0; d = 3; count = 0
2. for k = i to n loops in the remaining loop nest

If column k in dependence matrix has p “<”s ($p > 1$) with known distances whose maximum is e, one or more “>”s, and the other entries in column k are only of form “>=” and “=”, and ($e < d$ or ($e = d$ and $count < p$))

then

j = k; d = e; count = p

endif

3. If j = 0 then no loop is alignable, move to next available technique
4. Otherwise
   (a) if temporary is not allocated yet, slice the array assignment into computation section, store section and alignment section, reduce the temporary size in that dimension
   (b) peel d iterations of the computation section at the beginning
   (c) peel d iterations of the store section and alignment section at the end
   (d) adjust subscripts in computation section with respect to loop index j, change j to $(j + d)$ in the subscripts
   (e) If the temporary size is reduced for the first time, append new alignment statements in the alignment section

From this algorithm, we can see that loop alignment can only reduce the size of the temporary in one dimension, but not more. After first time it is applied to reduce the temporary, subsequent loop alignments can only prevent using the failsafe method which will split the computation section in the first loop nest, the store section and alignment section in the second nest. This can be seen in the 9-point stencil computation example as the appendix of this paper.

3.3 Loop alignment versus input prefetching

Input prefetching method is also targeted at the situation when a loop carries both true dependences and anti dependences. Which method should we use, loop alignment or input prefetching? Based on the following analysis, we can see that loop alignment is more powerful than input prefetching.

1. For every “<” entry in the current column, input prefetching needs to be applied to correct it. However, one loop alignment will correct all these entries. It is possible that these multiple prefetches can be merged into one, but doing so requires the analysis of the shape of the array reference subscripts in inner dimensions, which makes the scalarization more complicated.
2. Once loop alignment is applied, all “=” entries in the current column are turned into “>”s. Therefore, these dependences can be eliminated from consideration when proceeding to the inner loops. Input prefetching does not have this additional benefit.

3.4 Updated scalarization algorithm with loop alignment

Finally, an updated scalarization algorithm with loop alignment is given below. Based on the above discussion about loop alignment and input prefetching, we take input prefetching out of the old scalarization algorithm.

1. Naively scalarize the array assignment into loop nest
2. Compute the dependence matrix
3. for i = 1 to n loops in the loop nest
   (a) If i carries no true dependences, continue
   (b) If i carries only true dependences but no anti dependences, apply loop reversal, continue
   (c) Search for candidate loop j in loop nest (i+1, n) such that j carries either no true dependences or only true dependences, interchange loop i and j, apply loop reversal if necessary, continue
   (d) Search for candidate loop j in loop nest (i, n) such that j only carries true dependences with known distances and one or more anti dependences, set the align distance to be the maximum of the distances of true dependences, interchange loop i and j, apply loop alignment, continue
   (e) Otherwise, we need to allocate an temporary array of the size of remaining loops, split the loop which is the failsafe option. If the temporary array has already been allocated by previous loop alignment, we simply put the computation section in the first loop nest, the store section and alignment section in the second loop. break

4 Loop Skewing

Loop skewing is a transformation that has been used to remap an iteration space so that wavefront parallelism lines up directly with a loop iteration [3]. This transformation involves two or more loops that usually all carry dependences. By creating a new loop which is a linear combination of two or more loops, loop skewing tries to make one loop dependence-free, which can be parallelized. The penalty of loop skewing is the varying loop bounds.

What loop skewing really manipulates is the dependence. By associating two or more loops together and creating a new loop, array references become coupled and thus delta test needs to be performed. Take cases of two loops for instance, the dependence entries for new loop is the result of offsetting one old loop to the other old loop. The formula of offsetting depends on the the formula how two loop indices are linearly combined. Here we will only give the formula for simple reference subscripts. Let the two loops be "I" and "J", by simple subscripts we mean that they only appear in the form of "$\vec{a} * I + \vec{c}$" or "$\vec{a} * J + \vec{c}$", where $\vec{a}$ and $\vec{c}$ are constants. Let the dependence matrix be D, the combination formula be "$j = n * I + J$". The dependence entries for new loop "j" is "$\vec{d_j} = n * \vec{d_I} + \vec{d_J}$". This can be easily derived from delta test. Initially we have $a * (J_2 + \Delta J) + c = a * J_1 + c$, now we have $a * (j_2 - n * I_2 + \Delta J) + c = a * (j_1 - n * I_1) + c$, notice that $I_2 + \Delta I = I_1$, we have $\Delta j = n * \Delta I + \Delta J$.

Understanding how loop skewing works, we can use loop skewing during scalarization to convert loop carried true dependences into loop carried anti dependences. Let's look at an example. Suppose we need to scalarize `A(3:N+2, 3:M+2) = A(1:N, 5:M+4) + A(5:N+4, 2:M+1) + A(3:N+2, 5:M+4)`, the naively scalarized code is as follows:

```
DO J = 1, M
 DO I = 1, N
  A(I+2, J+2) = A(I, J+4) + A(I+4, J+1) + A(I+2, J+4)
 ENDDO
ENDDO
```

The scalarization dependence matrix is:

$$\begin{bmatrix} J & I \\ >_{-2} & <_{2} \\ <_{1} & >_{-2} \\ >_{-2} & =_{0} \end{bmatrix}$$

If we choose formula $j = I + J$ for loop skewing, we have

```
DO I = 1, N
 DO j = I+1, I+M
  A(I+2, j-I+2) = A(I, j-I+4) + A(I+4, j-I+1) + A(I+2, j-I+4)
 ENDDO
ENDDO
```

Move j loop to the outmost position, we have

```
DO j = 2, N+M
 DO I = max(1, j-M), min(N, j-1)
  A(I+2, j-I+2) = A(I, j-I+4) + A(I+4, j-I+1) + A(I+2, j-I+4)
 ENDDO
ENDDO
```

The dependence matrix now becomes

$$\begin{bmatrix} j & I \\ =_{0} & <_{2} \\ >_{-1} & >_{-2} \\ >_{-2} & =_{0} \end{bmatrix}$$

We see that I loop still carries a true dependence. This can be corrected by loop reversal, which will give the correct scalarized code.

```
DO j = 2, N+M
 DO I = min(N, j-1), max(1, j-M), -1
  A(I+2, j-I+2) = A(I, j-I+4) + A(I+4, j-I+1) + A(I+2, j-I+4)
 ENDDO
ENDDO
```

Actually, if the formula $i = I + J$ is used, "i" and "J" loops will only carry anti-dependences. Thus loop skewing itself is enough for generating the correct scalarized code. In the above example, we used the $j = I + J$ formula to demonstrate that loop skewing can be combined with other methods such as loop reversal to achieve a correct scalarization.

The benefit of loop skewing is to avoid the use of temporary array or at least reduce the dimensions of temporary array allocated when combining with other methods. In our example, if loop alignment is applied first, a temporary array of size $3 * M$ or $2 * N$ will have to be allocated, while loop skewing eliminates this space requirement.

However, to determine whether loop skewing is applicable, we need to search all the combinations of two or more loops in the remaining loop nest in the worst case. The desirable goal of loop skewing in the context of scalarization is to obtain a dependence matrix in one of the following three cases:

1. All entries in the new loop column are “>”s.

2. All entries in the new loop column are “<”s. If we reverse the new loop, they will become anti dependences.

3. All entries in the new loop column are “>”s and “=”s, or “<”s and “=”s. For the “=” rows, the entries in the other loop column are all “>”s and “=”s, or all “<”s and “=”s.

From this goal, we can see that the searching space is exponential to the number of loops in the scalarized loop nest. This makes it not easy to solve the equations in order to obtain the loop skewing formula. Analysis will be more complicated when more than two loops are involved or array reference subscripts are not simple. Fortunately, the number of dimensions of arrays is usually a small number in practice, and we only treat the loops that carry dependences as skewing candidates. Compared to other techniques mentioned above, loop skewing incurs more expensive analysis.

5 Experiments

We implemented loop alignment in the D system developed at Rice University. Previous scalarization code already has loop reversal and loop interchange. We didn't implement loop skewing due to the complexity of the associated analysis. The tool we built is basically a source to source transformer. Given a Fortran 90 program with array syntax inside, our tool will only scalarize array syntax into loop nests, while keep other parts of the program unchanged. This transformed program is then passed to the commercial compilers for code generation.

The code we run experiments on is a Jacobi iterative method for solving Partial Differential Equations (PDE). It implements a nine-point stencil. The major computation takes part in the following array assignment statement,

```
grid(2:n+1, 2:n+1)  = (grid(1:n, 1:n) + grid(1:n, 2:n+1) + grid(1:n, 3:n+2)
                    + grid(2:n+1, 1:n) + grid(2:n+1, 2:n+1) + grid(2:n+1, 3:n+2)
                    + grid(3:n+2, 1:n) + grid(3:n+2, 2:n+1) + grid(3:n+2, 3:n+2))
                    / 9.0
```

And this statement is iterated for 400 time. To scalarize this statement, without loop alignment, a full size temporary array has to be generated. The right hand side is computed into the temporary array first, followed by copying the result from the temporary back into the `grid`. People are aware of it. In Compaq Fortran memory management tutorial, they actually allocated two copies of the grid called new and old, at the end of the computation, the new and old are switched. By doing so, the copying operation is eliminated. However, if loop alignment is used, we can reduce the temporary size to $2 * n$.

We tested 3 versions of the Jacobi code, the first one with whole size temporary array and copying operation, the second one with doubled array size, no copying operation which is saved by the switching operation, the last one with loop alignment applied. We want to show that by saving the size of arrays, the memory hierarchy performance can be improved. These three versions of programs are run on a SGI O2 for testing running time, TLB misses, primary cache misses and secondary cache misses. The size of the grid are set to be 100x100, 200x200, 400x400, 800x800, and 1600x1600. The measurements are shown in the following figures.

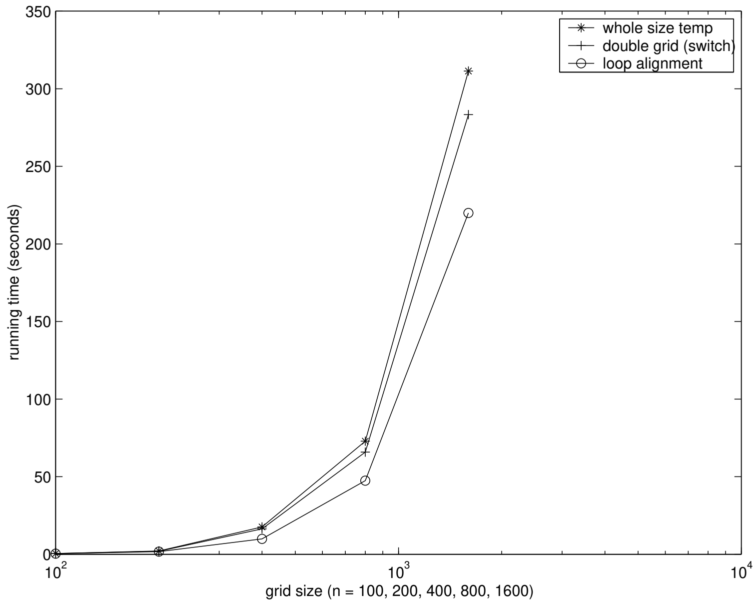

Figure 1: Running time

From Figure 1, we can see that the running time of the reduced temporary version is less than the other two versions. This matches what we have expected. Now we need to examine if the memory hierarchy contributed to this running time gain. The TLB misses of loop alignment version are generally less than those of the other two versions as shown in Figure 2. In Figure 3, the primary cache misses show the similar pattern until the grid reaches a large size (1600x1600). However, when the grid size gets larger, the secondary cache misses become the dominant factor. From Figure 4, we can see that the secondary cache misses of loop alignment version is 50% less than the other two version for large grid sizes.

We also tried to find other applications as examples but not very successful. Firstly, most iterative PDE solvers need to check convergence after each iteration, therefore a copy of the old array has to be kept for computing the difference. Secondly, we conjecture that application programmers are probably aware of the temporary array that will be allocated. They choose to write hand optimized code in loop nests instead of using array syntax.

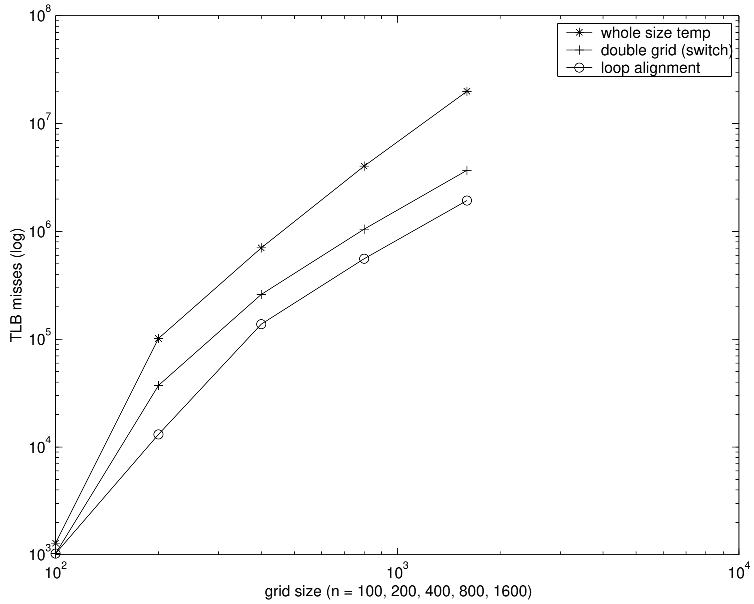

Figure 2: TLB misses

6 Conclusion

In this paper, we have presented two new techniques for advanced scalarization. The goal of these two techniques is to eliminate the need of a temporary array or reduce the size of the temporary, therefore, improve the memory hierarchy performance on a single scalar machine by reducing the memory bandwidth requirement. We have implemented loop alignment and our experiments have demonstrated the dramatic performance gain on large data sets. Although we didn't implement loop skewing due to the complexity of analysis, we can expect a similar performance gain as loop alignment or even slightly better.

The scalarization techniques can be applied to other languages with array syntax features, though our experiments were with Fortran 90. For example, the POOMA (Parallel Object-Oriented Methods and Application) library has an interval notation which has similar semantics to array section in Fortran 90.

Both loop alignment and loop skewing handles the case where naively scalarized loop nest carries both true and anti dependences. By reducing the size of temporaries, these two techniques may prohibit the parallelism that can be achieved by using the failsafe scalarization method. How to parallelize array assignment while maintaining efficient memory hierarchy performance remains as future study.

Another direction we didn't address in this paper is to improve memory hierarchy performance across different array assignment statements by loop fusion. As many paper suggested, the analysis should be conducted at the array assignment statement level [5, 6, 7, 8]. We need to study the profitability of loop fusion which improves data reuse versus loop alignment and loop skewing since these two transformations deform the iteration space, which may prevent effective fusion.

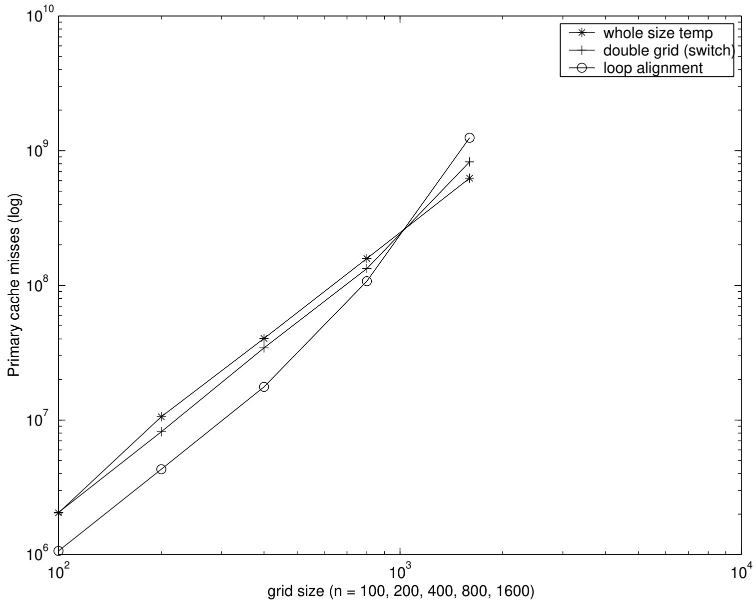

Figure 3: Primary cache misses

7 Appendix: 9-point Stencil Example

A sample program written in array syntax is as follows:

```
program stenciltest
  use size             ! where N gets defined
  implicit none
  integer I, J
  real c(N, N), sum

  ! ... Initialization omitted here

  DO I = 1, 9999
    c(2:N-1, 2:N-1) = (c(1:N-2, 1:N-2) + c(1:N-2, 2:N-1) + c(1:N-2, 3:N) &
                     + c(2:N-1, 1:N-2) +                 c(2:N-1, 3:N) &
                     + c(3:N, 1:N-2) + c(3:N, 2:N-1) + c(3:N, 3:N) &
                         ) / 8.0
  ENDDO

  ! ... output code omitted
end
```

Output code coming out of our scalarizer is as follows:

```
      program stenciltest
        use size
        implicit none
```

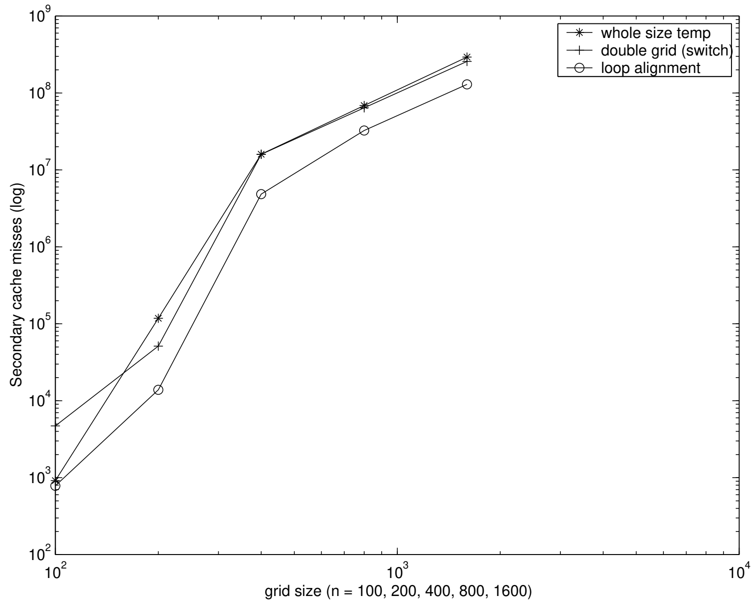

Figure 4: Secondary cache misses

```
      integer i, j
      real c(n, n), sum
!     ------------------------------------
!     Temporaries for naive vectorizations
!     ------------------------------------
      integer id$1tmp, id$0tmp
!     -----------------------------------
!     Temporaries for smart vectorization
!     -----------------------------------
      real a$tmp2
      dimension a$tmp2(:, :)
      allocatable a$tmp2
!
!

!     ! ... Initialization omitted here

      do i = 1, 9999
        allocate (a$tmp2(n - 2, 2))
        do id$1tmp = 1, 1
          do id$0tmp = 1, n - 2
            a$tmp2(id$0tmp, id$1tmp) = (c(id$0tmp, id$1tmp) + c(id$0tm
     *p, id$1tmp + 1) + c(id$0tmp, id$1tmp + 2) + c(id$0tmp + 1, id$1tmp
     *) + c(id$0tmp + 1, id$1tmp + 2) + c(id$0tmp + 2, id$1tmp) + c(id$0
     *tmp + 2, id$1tmp + 1) + c(id$0tmp + 2, id$1tmp + 2)) / 8.0
          enddo
        enddo
```

```
      do id$1tmp = 1, n - 3
        do id$0tmp = 1, 1
          a$tmp2(id$0tmp, 2) = (c(id$0tmp, id$1tmp + 1) + c(id$0tmp,
* id$1tmp + 1 + 1) + c(id$0tmp, id$1tmp + 1 + 2) + c(id$0tmp + 1, i
*d$1tmp + 1) + c(id$0tmp + 1, id$1tmp + 1 + 2) + c(id$0tmp + 2, id$
*1tmp + 1) + c(id$0tmp + 2, id$1tmp + 1 + 1) + c(id$0tmp + 2, id$1t
*mp + 1 + 2)) / 8.0
        enddo
        do id$0tmp = 1, n - 3
          a$tmp2(id$0tmp + 1, 2) = (c(id$0tmp + 1, id$1tmp + 1) + c(
*id$0tmp + 1, id$1tmp + 1 + 1) + c(id$0tmp + 1, id$1tmp + 1 + 2) +
*c(id$0tmp + 1 + 1, id$1tmp + 1) + c(id$0tmp + 1 + 1, id$1tmp + 1 +
* 2) + c(id$0tmp + 1 + 2, id$1tmp + 1) + c(id$0tmp + 1 + 2, id$1tmp
* + 1 + 1) + c(id$0tmp + 1 + 2, id$1tmp + 1 + 2)) / 8.0
          c(id$0tmp + 1, id$1tmp + 1) = a$tmp2(id$0tmp, 1)
          a$tmp2(id$0tmp, 1) = a$tmp2(id$0tmp, 2)
        enddo
        do id$0tmp = n - 2, n - 2
          c(id$0tmp + 1, id$1tmp + 1) = a$tmp2(id$0tmp, 1)
          a$tmp2(id$0tmp, 1) = a$tmp2(id$0tmp, 2)
        enddo
      enddo
      do id$1tmp = n - 2, n - 2
        do id$0tmp = 1, n - 2
          c(id$0tmp + 1, id$1tmp + 1) = a$tmp2(id$0tmp, id$1tmp - n
*+ 3)
        enddo
      enddo
      deallocate (a$tmp2)
    enddo

    ! ... output code omitted$
  end
```

References

[1] Adams, Jeanne C., Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener, *The Fortran 90 Handbook*, McGraw-Hill, 1992.

[2] J. R. Allen, *Dependence Analysis for Subscripted Variables and its Application to Program Transformation*, PhD thesis, Rice University, Houston, TX, 1983.

[3] R. Allen and K. Kennedy, *Advanced Compilation for High Performance Computers*, Morgan Kauffman, to be published.

[4] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams, *Programmer's Guide to Fortran 90*, McGraw-Hill, 1990.

[5] 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.

[6] 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.

[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, *Proceedings of the 9th International Conference on Compiler Construction (CC'2000)*, March 2000.

[9] D. Wedel, FORTRAN for the Texas Instruments ASC system, *SIGPLAN Notices* 10(3), March 1975, 119-132.

[10] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, PhD Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, November, 1982.