A Case Study of Some Issues in the Optimization of Fortran 90 Array Notation

JOHN D. McCALPIN

The Graduate College of Marine Studies, University of Delaware, Newark, DE 19716-3501; e-mail: mccalpin@udel.edu

ABSTRACT

Some issues in the relationship of coding style and compiler optimization are discussed with regard to Fortran 90 array notation. A review of several important Fortran 90 array constructs and their performance on vector and scalar hardware sets the stage for a more detailed example based on the kernel of a finite difference computational fluid dynamics model, specifically the nonlinear shallow water equations. Special attention is paid to the optimization of memory use and memory traffic. It is shown that the style of coding interacts with the rules of Fortran 90 and the current state of the art of Fortran 90 compilers to produce a fairly wide range of performance levels. Although performance degradations are typically small, a few cases of more serious loss of effciency are identified and discussed.

1 INTRODUCTION

The array notation provided by Fortran 90 can, if used carefully, provide significant benefits in the readability and reliability of scientific codes. Less commonly understood are the performance implications of the specification of algorithms in the Fortran 90 array notation. This article presents several examples of code segments written in Fortran 77 and in various styles of Fortran 90 array notation, and explains how certain aspects of the optimization problem for the Fortran 90 code are significantly more difficult than for the Fortran 77 code. These differences are typically greatest for hierarchical memory computers (i.e., those with caches), but apply to vector architectures as well.

Received January 1995
Revised July 1995

Scientific Programming, Vol. 5, pp. 219–237 (1996)
CCC 1058-9244/96/030219-19

The Fortran 90 array semantics make some aspects of vectorization and parallelization much simpler—the compiler can assume no data dependencies exist within a single array statement, so the calculations for that statement can be performed in any convenient order. Some early commentators suggested that this would, in itself, justify the transition to Fortran 90. An example is [1], which considers the data-dependency aspects of the translation of Fortran 77 code to Fortran 90 array notation. Unfortunately, the characteristics of high-performance computers have changed significantly since the Fortran 90 array notation was originally considered. First, vectorization is no longer a dominating issue—powerful vectorizers exist for vector and pipelined hardware already. Second, computer floating-point performance continues to advance much more rapidly than memory performance, with the fastest current central processing units (CPUS) capable of performing several hundred floating-point operations in the time required to service a single data cache miss. Therefore, optimization of data reuse (i.e., optimi-

zation *between* statements rather than *within* statements) is now the crucial issue in high-performance computing on hierarchical memory systems (and it remains an important issue for nonhierarchical systems). In this article, I examine the interaction of coding style with the performance of two of the better currently available Fortran 90 compilers with specific reference to the optimization of memory traffic, and conclude with some suggestions for coding style, future compiler development, and language standard development.

2 OPTIMIZATION ISSUES

In the following examples, the pieces of optimization technology that are of potential relevance are loop fusion, array-valued common subexpression elimination, reduction of rank of implied array temporaries, and function inlining. These optimizations and many others are discussed in a recent comprehensive review [2], but are addressed here in the specific context of Fortran 90 array notation.

2.1 Array Temporaries

To introduce some of the issues involved in the compilation of Fortran 90 array notation, I will begin with the simplest of expressions:

```
* Example 1a
      REAL A(M),B(M),C(M),Q
      DO I=1,M
         A(I)=B(I)+Q*C(I)
      END DO
```

and its Fortran 90 counterpart

```
! Example 1b
REAL A(M),B(M),C(M),Q
A=B+Q*C
```

The essential difference between these two statements of the vector operator is that the Fortran 90 semantics guarantee that the results are independent of the order of the calculation of the elements of the arrays. Thus, if data dependencies exist between the right and left sides of an expression, the Fortran 90 semantics imply the use of a temporary array to hold the result, so that the target array is not modified until all calculations are complete. The "safe" way to compile this structure is

```
! Example 1c
ALLOCATE (TMP(1:M))
DO I=1,M
   TMP(I)=B(I)+Q*C(I)
END DO
DO I=1,M
   A(I)=TMP(I)
END DO
DEALLOCATE (TMP)
```

where TMP is dynamically allocated temporary array. This implementation of the code requires $3M$ loads, $2M$ stores, M multiply-add arithmetic operations, and two library calls, while the original Fortran 77 code requires $2M$ loads, M stores, and M multiply-add arithmetic operations. On hierarchical memory machines with a *write-allocate* cache policy ([13]), each write will result in the reading of another cache line containing the data to be overwritten. (At least one early implementation of Fortran 90 took this idea to an extreme and separated the add and multiply into separate loops. Since the evaluation of the right-hand side by itself does not modify any user-defined arguments, it is clear that these two operations can always be safely combined.)

The expected slowdown ratio on a write-allocate, hierarchical memory machine can be estimated as the ratio of the memory traffic, or 7 : 4 for uncached operands. On a vector computer, the ratio is estimated as 5 : 3. The actual performance of version 1c vs. version 1a is presented in Figure 1, as a function of the base 2 log of the size (in 64-bit words) of each array. Results are included

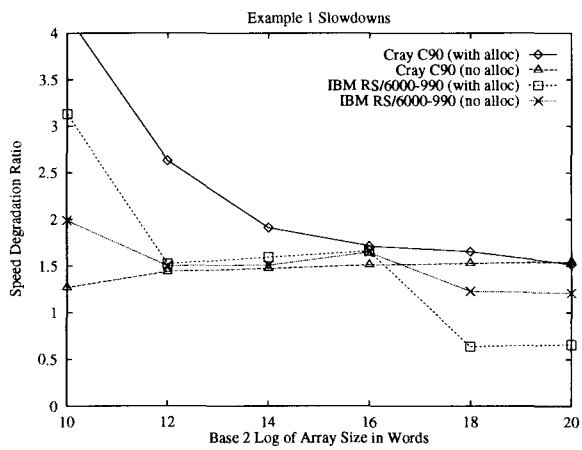

FIGURE 1 Slowdown ratio as a function of problem size due to the addition of a temporary array to a linked triad vector operation. Curves marked "no alloc" exclude the cost of allocating the temporary array.

both for statically and dynamically allocated versions of the TMP array, for array sizes from 1K words to 1M words. The results presented are the minimum slowdown observed in tests with array offsets ranging from M to $M + 7$ words, and five repetitions for each case. For small arrays, the overhead of the temporary array allocation is high, resulting run-times tripled or worse, and can be expected to grow much higher for smaller arrays (which were not included here because of the difficulty of obtaining accurate timings). For intermediate array sizes (and for cases in which the array allocation cost is excluded), the ratios are close to 1.5 in all cases, which is a bit better than expected. An unexpected result occurred on the IBM RS/6000 for large array sizes, in which two-loop coding with a temporary array actually outperformed the single-loop coding. The tests were performed with a variety of array alignments and this result was found to be robust at the "-O2" level of optimization. I have no explanation for the improved behavior of version 1c, and assume that it is due to TLB thrashing, since it only occurs for cases where the TLB addressing range of 2 MB is exceeded. At the "-O3" level of optimization, the loops were recombined by the compiler (since there is, in fact, no data dependence in this example), yielding no performance degradation.

Of course, the data-dependence analysis required to determine that no temporary arrays are needed in the preceding case is not challenging (the compiler has to check to be sure that the arrays are no aliased via EQUIVALENCE statements). However, the same principal applies to more complex codes (and to some extent to codes based on Fortran 90 POINTER variables) where the data dependence can be either difficult or impossible to analyze. A slightly more complex example is

```
! Example 2a
REAL U(M,2)
INTEGER NEW,OLD

U(2:M-1,NEW)=U(3:M,OLD)-U(1:M-2,OLD)
```

which corresponds to the Fortran 77 loops

```
* Example 2b
      DO I=2,M-1
         TMP(I)=U(I+1,OLD)-U(I-1,OLD)
      END DO
      DO I=2,M-1
         U(I,NEW)=TMP(I)
      END DO
```

In this case, a data dependence exists (and thus a temporary array is required) only if NEW is equal to OLD. If the compiler is not able to determine whether NEW.EQ.OLD, then it is faced with two choices: Code the loop the safe way with a temporary or code two versions of the loop with an explicit test comparing NEW and OLD. The user, on the other hand, is only able to provide information to the compiler by use of compiler directives or by moving the calculations to subroutines (and making use of the Fortran requirement that subroutine arguments be free of aliasing). The former route is not standardized, while the latter is obscure in both presentation and intent.

In testing the code above, I found no compilers that split the code into two cases for this particular test—all used a temporary array to hold the intermediate values. Creating multiple versions of a loop to enable vectorization is part of current practice [3], but building multiple versions to eliminate temporaries is apparently not. Without the temporary array, the code requests $2M$ loads and M stores, while with the temporary array, the code specifies $3M$ loads and $2M$ stores. On a cached machine, the two terms on the right-hand side usually lie in the same cache line, so the loop with no temporary requires (effectively) M loads and M stores, while the loop with a temporary requires $2M$ loads and $2M$ stores, so that we expect asymptotic speed reduction by a factor of 2.0.

On the IBM RS/6000, the actual results were somewhat variable, depending on the size and details of the problem. The use of a temporary in the loop above slowed down the execution by factors of close to 2 (provided that the ALLOCATE could be hoisted out of the timing repetition loop), while some smilar loops (for which the temporary array fit into the data cache) were slowed down by as little as 33%. Interactions of this extra temporary array with the translation lookaside buffer and the associativity of the cache further complicate the problem of estimating the performance impact of failing to remove the extra array when it is not needed. On the Cray Research vector machines, the performance degradation due to the extra temporary is a factor of approximately 1.35, and is not significantly altered by problem size.

Both the Cray C90 and IBM RS/6000-990 are characterized by good memory bandwidth compared to peak CPU performance. To illustrate the effect of the extra temporary on a machine with a much lower ratio of main memory bandwidth to cache bandwidth and a much larger cache, the performance of example 2 on an SGI Power Chal-

lenge is presented in Figure 2. The slowdown ratio ranges from a factor of about 1.7 (when all loops are cache contained) to a factor of 8 (when the addition of a temporary causes the loop to exceed the cache size), and finally to a limiting value of approximately 2.5 (when the arrays are all larger than cache). The factor of 8 is very close to the ratio of sustainable main memory bandwidth to cache bandwidth measured independently. Again, all of the above measurements were repeated several times, and with array offsets varied to ensure that no pathological cache associativity or memory bank conflict problems were present. These results are likely to be typical for other hierarchical memory machines with low sustainable main memory bandwidth relative to cache bandwidth, such as the PA-RISC machines from Hewlett-Packard and the "Alpha"-based machines from Digial Equipment Corporation.

2.2 Loop Fusion

When writing Fortran 77 codes, experienced users typically combine adjacent loops if they are conformable and contain no data dependencies. This optimization is typically referred to as "loop fusion" or "jamming" [2]. Like "loop unrolling," loop fusion was originally envisioned as a means of reducing loop overhead. Although reducing loop overhead is still an issue for some processor families, a more important factor for performance is that loop fusion may allow data reuse in the combined loop, thus reducing loads and stores to main memory. As memory accesses become more costly relative to CPU operations, such fusion will become more and more necessary and commonplace.

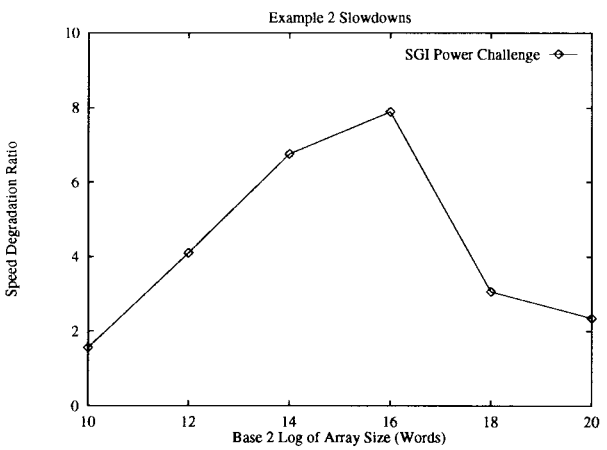

FIGURE 2 Slowdown ratio as a function of problem size due to the addition of a temporary array to a potentially data-dependent operation. Results are for a Silicon Graphics Power Challenge, and do not include the cost of allocating the temporary array.

With current compiler technology, loop fusion requires that the loops be conformable, provably non-data dependent, and adjacent (although separation by scalar references may be ignored in some cases). The user should be aware of each of these issues when trying to understand what a Fortran 90 compiler is doing to optimize a piece of code.

Consider a problem (an example of which will appear below in Section 3), for which the arrays are naturally seen to be composed of "interior" and "boundary" elements. This distinction is associated with different calculations on the two types of points. A natural specification of the problem might be to define the interior calculations only for the interior points (using the triad array section notation) and then to define the boundary calculations to complete the array update.

```
! Example 3a
REAL, DIMENSION(N,N) :: A,B,C

A(2:N-1,:)=C(2:N-1,:)-B(2:N-1,:)
A(1,:)=0.0
A(N,:)=0.0

B(:,2:N-1)=A(:,2:N-1)+C(:,2:N-1)
B(:,1)=1.0
B(:,N)=1.0

! etc....
```

Because the two-dimensional array sections are neither conformable nor adjacent, current compilers will not be able to make any useful fusion of the above loops. (The boundary loops for A can be fused, as can the boundary loops for B, but this does not allow any data reuse and so will be of negligible performance impact.) However, it is relatively simple to change the code to perform the interior calculations at each point of each array, and then subsequently to correct the edges of each array to allow more effective fusion, as in Example 3b.

```
! Example 3b
REAL, DIMENSION(N,N):: A,B,C

A=C-B       ! assign at all points, even though
B=A+C       ! the boundaries will be incorrect

A(1:N)=0.0 ! fix the boundaries
A(N,:)=0.0

B(:,1)=1.0 ! fix the boundaries
B(:,N)=1.0
```

On the IBM RS/6000, the code of Example 3a took 77% longer to execute than the code of Example 3b for large array sizes, while on the Cray C90, Example 3a took fully three times as long as Example 3b. On both systems, the compilers collapsed the code as much as possible.

In an optimization called "loop peeling" [2], the compiler may remove some iterations from the beginning or ending of a loop to enable vectorization or fusing. A generalization of this optimization could also be applied in Example 3. Because the optimization *removes* iterations from the loop (rather than *adding* them as I did), it will result in more "fixing" of the boundaries than Example 3b required. It does not appear that multidimensional versions of loop peeling (which, along with array statement reordering, would be required for Example 3a) exist as part of current compiler technology, although some development in that direction can be expected in the future.

2.3 Array-Valued Functions and Inlining

Array-valued functions are a very expressive feature of Fortran 90 array notation, and can be quite useful in simplifying the presentation of an algorithm. Several features of array-valued functions should be kept in mind when analyzing their performance.

First, the initial generation of Fortran 90 compilers does not include any (that I am aware of) that inline array-valued functions. Lack of inlining means that the use of array-valued functions will typically inhibit loop fusion, even if the array references inside the function are conformable with the result. If they are not conformable, the use of array-valued functions may prevent the user from doing the rearrangements illustrated in the previous section to assist the compiler.

Second, the Fortran language allows functions to have side effects in many circumstances. If the compiler is not able to prove that a function has no side effects, then it will not be possible to perform common subexpression elimination on that reference (such as automatically converting two identical array-valued function references into one reference with the result saved in temporary array). Separate compilation is more likely to result in conservative choices by the compiler, and internal (CONTAIN'ed) functions are more likely to be effectively analyzed for data dependence and side effects. In either case, it is likely that compiler directives at the point of reference will be necessary to allow common subexpression elimination on array-valued functions.

2.4 Optimization of Array Temporaries and Common Subexpression Elimination

At the beginning of this section, I introduced the basic idea of array temporaries and showed that they are introduced when there is a real or potential data dependence between the RHS and LHS of an array statement. The use of temporaries is always safe (i.e., the code will generate the correct answer whether or not a data dependence is actually present), and the data-dependence analysis required to remove them can be quite difficult. The question of efficiency is considerably more difficult. Consider a set of consecutive two-dimensional loops with a potential data dependency on one of the two indices:

```
! Example 4a
SUBROUTINE FOURA (A,B,C,N,M,OLD,NEW)
REAL, DIMENSION(N,M,2):: A, B, C
INTEGER OLD,NEW
```

```
A(2:N-1,:,NEW) = (A(3:N,:,OLD)-A(1:N-2,:,OLD)) &
                 +(B(3:N,:,OLD)-B(1:N-2,:,OLD))
B(2:N-1,:,NEW) = (B(3:N,:,OLD)-B(1:N-2,:,OLD)) &
                 +(C(3:N,:,OLD)-C(1:N-2,:,OLD))
C(2:N-1,:,NEW) = (C(3:N,:,OLD)-C(1:N-2,:,OLD)) &
                 +(A(3:N,:,OLD)-A(1:N-2,:,OLD))
END
```

Because of the potential data dependencies on the first index, the compiler will generate a temporary array for each statement (although it is possible to make multiple versions based on a comparison of OLD and NEW and generate code without temporaries when they are unequal). To reduce the rank of the temporary array from two dimensions to one, the compiler must recognize that there is no potential data dependence on the second index in this example. Fusion of the inner loops is inhibited because of potential data dependencies between the statements. This inhibition could also occur if the array statements were not conformable.

The real difficulties are then in the next step of deciding what to do to optimize those temporaries, and to decide whether or not to make use of the common subexpressions on the right-hand sides of the three array statements. Assuming that the compiler has already reduced the rank of the implied temporary from two dimensions to one, we still must consider the following two extremes:

1. N is small: The temporaries are each small (N-2 elements), and the overhead of allocation is relatively large. The allocations and deallocations should be hoisted out of the J loop entirely, and occur only once per subroutine invocation. The common subexpressions need only be evaluated once and saved in other temporary arrays.
2. N is very large: The temporaries are each large, and allocating all three at once would result in too much memory usage. Each array should be allocated and deallocated separately for each invocation of each inner loop so that the total temporary space does not exceed N-2 elements. The allocation of additional temporary arrays to optimize the common subexpressions would also result in too much memory usage.

Because the temporary arrays are conformable in this example, it is conceivable that the compiler could collapse them into a single temporary, the allocation of which could be safely hoisted out of the J loop. This optimization does not appear to be part of current practice, and may confuse later stages of optimization by making the active range of the values in the temporary array less clear. (The ALLOCATE and DEALLOCATE statements provide a very clear message that data in the array do not need to be saved between loops.) Temporary arrays to hold common subexpressions cannot be collapsed in the same way, because they are needed to propagate information between statements.

Note that there is no way for the compiler to be able to judge between these choices, because the limit of "too much memory" depends on configuration and load of the target computer, as well as the presence or acceptability of the use of virtual memory. The "safe" choice of allocating and deallocating as soon as possible, and not attempting array-valued common subexpression elimination, will typically result in decreased performance, particularly for small arrays. The user must either accept the strategy chosen by the compiler (which may easily differ from system to system), change the code to manually control the temporary arrays, or provide extra hints to the compiler via directives.

A final minor point is that common subexpression elimination in Example 4a is typically of no benefit on hierarchical memory machines unless the loops are fused. If the arrays are larger than cache, the cost of loading the temporary array with the common subexpression is almost the same as the cost of loading the two array elements (which, being contiguous, will result in no more cache misses than loading the temporary array) and performing the subtraction. If the arrays are almost as large as cache, the use of the additional temporary array might cause cache thrashing. Finally, if the common subexpressions had their data dependency on the second index instead of the first, then the common subexpression elimination would again be potentially useful, as the re-evaluation of the expression would require twice as many cache misses as loading the temporary array.

2.5 Masked Array Assignments—The WHERE statement

The WHERE statement provides a very elegant and general means of specifying conditional operations. For example, the Fortran 90 statement

```
REAL, DIMENSION(N):: A,B,C
WHERE (A.GT.B) C = A-B
```

is equivalent to the Fortran 77 code

```
REAL A(N),B(N),C(N)
LOGICAL L(N)
DO I = 1,N
   L(I) = (A(I).GT.B(I))
END DO
DO I = 1,N
   IF (L(I)) C(I) = A(I)-B(I)
END DO
```

I have included the temporary logical array in the above to emphasize the Fortran 90 requirement that there be no data dependence between the mask (A.GT.B) and the assignment. Such a dependence could exist, for example, if the LHS of the assignment were used as part of the logical expression, or if the logical expression involved an array-valued function call that could have a side effect of modifying the right-hand side of the assignment.

It is clear from the Fortran 77 equivalent code that the WHERE statement can result in both increased data motion and degraded arithmetic performance relative to the unconditional expressions, by (in effect) placing an IF statement inside an inner DO loop, which is an inefficient construct on most hardware. Fortunately, the WHERE statement can be applied to blocks of code, in which case the cost of the logical operation and the associated extra memory traffic may be reduced compared to the cost of the arithmetic expression(s) to be performed. Because the statements within the WHERE block must be conformable, the important optimization issue is then identifying data dependence between statements in the WHERE block that would inhibit loop fusion.

A special case occurs when (1) the same array expression is used as a mask for many operations, and (2) the geometrical connectivity of the TRUE values of the mask is relatively simple. In such cases, it is generally faster to preprocess the mask to determine the starting and stopping array indices for each row of the array. For the case of a simply connected set of TRUE values of the mask, the Fortran 77 code would look something like:

```
DO J=JSTART,JSTOP
   DO I=ISTART(J),ISTOP(J)
      C(I) = A(I)-B(I)
   END DO
END DO
```

In this case, there are no longer any logical operations in the inner loop. In addition to making scheduling easier, the mask array no longer needs to be loaded (assuming that it had to be calculated independently of the expression(s) in the WHERE block), thus potentially reducing the demands on the memory system.

2.6 Summary

The key to most of the optimizations discussed in the preceding sections is conformability. Adjacent, conformable array statements are much more likely to be amenable to the optimizations of common subexpression elimination, loop fusion, and overlapping of array temporaries. For the current generation of compilers, manual administration of temporary arrays may provide better performance than automatic administration, both because the allocation can be hoisted more effectively by the user, and because the user should be able to make better judgments regarding overlapping and reuse of temporary arrays.

3 AN EXAMPLE APPLICATION

In this section I will provide a larger example illustrating the principles presented in the previous section. The code is a finite difference discretization of the two-dimensional nonlinear shallow-water equations in a rotating coordinate system, and is a generalization of the "CHANNEL" benchmark program which I contributed to the Quetzal Fortran 90 benchmark suite [9, 12]. The "CHANNEL" benchmark, in turn, grew out of experiences with a linear shallow-water model originally written in Fortran 90 in support of a study of wave scattering in a slowly varying medium [10]. When the performance of the original Fortran 90 coding/compiler combination proved inadequate, a vari-

ety of experiments were undertaken to improve the performance, and the results are presented here.

For reference, and for comparison with the code, the governing partial differential equations (with subscripts denoting derivatives) are:

$$u_t + uu_x + vu_y - (f_0 + \beta y)v = -gh_x \quad (1)$$

$$v_t + uv_x + vv_y + (f_0 + \beta y)u = -gh_y \quad (2)$$

$$h_t + uh_x + vh_y + h(u_x + v_y) = 0. \quad (3)$$

subject to no flow through the boundaries at $x = 0$ and $x = L_x$, and linearized characteristic boundary conditions at $y = 0$ and $y = L_y$:

$$u = 0 \text{ at } x = 0 \quad (4)$$

$$u = 0 \text{ at } x = L_x \quad (5)$$

$$h + \frac{\sqrt{H}}{g} v = \text{specified at } y = 0 \quad (6)$$

$$h - \frac{\sqrt{H}}{g} v = \text{specified at } y = L_y \quad (7)$$

where H represents a time-independent mean value of h.

The equations are discretized with standard centered (explicit) second-order differences in space and time. In the code to follow, the indices I and J will represent spatial locations on the grid, while the third index of the array indicates the time level of the time integration scheme. This last index is shuffled as the time counter is incremented to fit the three time levels needed for the calculation into an array with a third dimension equal to three.

It is, of course, not important to understand the details of either the preceding equations or the following codes, but several items should be noted:

1. The variables u, v, and h are reused many times on the right-hand sides of the equations.
2. The time integration scheme introduces no data dependencies among the calculations and assignments constituting one time step.
3. The boundary conditions are applied at different locations for each variable, so the range of grid indices over which the equations are applied differs among the variables.
4. The derivative quantities u_x, v_x, h_x, h_y are each used twice in the calculations of the right-hand sides.
5. Assuming that the most obvious loop-independent operations are optimized out of the loops, there are approximately 40 floating-point operations per grid point per time step.

There are a variety of approaches to coding this in Fortran 77, and even more in Fortran 90. I make no claim to completeness, but instead present examples that have come from my own attempts to write Fortran 90 code in a clear fashion. Some of the comparisons below are unfair—such as comparisons between naively coded Fortran 90 and carefully optimized Fortran 77. This is deliberate, and is based on two motivating factors. First, most users of Fortran 90 are still "naive" Fortran 90 programmers, having had relatively little time to gain experience with issues of optimization and language style. Second, I anticipate that few scientific users are interested in porting their codes to Fortran 90 without some benefit—and that benefit is likely dominated by the perception that Fortran 90 is a route to more clear, reliable, and maintainable code. My goal is not to criticize Fortran 90 or its implementations, but to make users more aware of the optimization issues involved, and how they interact (sometimes badly) with the users' coding style.

Four example codes are presented and analyzed:

1. Example 5a—"Array Functions" uses array functions for the derivative operators.
2. Example 5b—"Explicit Temps" precalculates the derivative operators and saves in static temporary arrays.
3. Example 5c—"Pointers" is a very clear presentation using array functions and pointers.
4. Example 5d—"Inline Fortran 77" contains inlined code for the derivative functions.

3.1 Example 5a—"Array Functions"

This basic implementation of the algorithm is very similar to my first Fortran 90 implementation of the code. The implementation is fairly concise, although not as easily readable as one might desire. Note that appropriate compiler options are chosen so that all REAL values correspond to 64-bit precision on all machines tested.

```
! Example 5a --- ''Array Functions''
INTEGER, PARAMETER::      M=500,N=500
REAL,DIMENSION(M,N,3)::   U,V,H
REAL,DIMENSION(M,N)::     F

!--- BEGIN TIME MARCHING LOOP ---
DO ISTEP = 1,NSTEPS
     OLD = MOD(ISTEP+2,3)+1
     MID = MOD(ISTEP+3,3)+1
     NEW = MOD(ISTEP+1,3)+1
     ! ------ INTERIOR CALCULATIONS ------ !
U(:,:,NEW) = U(:,:,OLD) - (2*DT) * (      &
              U(:,:,MID)* D_DX(U(:,:,MID)) &
             +V(:,:,MID)* D_DY(U(:,:,MID)) &
             -F(:,:)*V(:,:,MID)
             +G* D_DX(H(:,:,MID)) )
V(:,:,NEW) = V(:,:,OLD) - (2*DT) * (      &
              U(:,:,MID)* D_DX(V(:,:,MID)) &
             +V(:,:,MID)* D_DY(V(:,:,MID)) &
             +F(:,:)*U(:,:,MID)            &
             +G* D_DY(H(:,:,MID)) )
H(:,:,NEW) = H(:,:,OLD) - (2*DT) * (      &
              U(:,:,MID)* D_DX(H(:,:,MID)) &
             +V(:,:,MID)* D_DY(H(:,:,MID)) &
             +H(:,:,MID)*                  &
              (D_DX(U(:,:,MID))+D_DY(V(:,:,MID))) )
    ! ------ BOUNDARY CALCULATIONS ------ !
    U(1,:,NEW) = 0 ; U(M,:,NEW) = 0
    V(:,1,NEW) = 0 ; H(:,1,NEW) = 0
    V(:,N,NEW) = 0 ; H(:,N,NEW) = 0
END DO
```

One of the array-valued functions used to evaluate the derivatives is shown below—the other is completely analogous, although its lack of conformability is one the second index rather than the first.

```
FUNCTION D_DX(ARRAY)
IMPLICIT REAL (A-H,O-Z)
REAL::       ARRAY(:,:), DX, C
REAL::       D_DX(SIZE(ARRAY,DIM=1,SIZE(ARRAY,DIM=2))
COMMON /PARMS/ DX

I = SIZE(ARRAY,DIM=1) ; J = SIZE(ARRAY,DIM=2)
C = 1.0D0/DX

D_DX(2:I-1,1:J) = C*(ARRAY(3:I,1:J)-ARRAY(1:I-2,1:J))

D_DX(1,1:J) = 2*C*(ARRAY(2,1,:J)-ARRAY(  1,1:J))
D_DX(I,1:J) = 2*C*(ARRAY(I,1:J)-ARRAY(I-1,1:J))

END FUNCTION
```

Notes

1. There are 10 references to array-valued functions in the inner loop. Four of the 10 references occur twice each, and the last 2 occur only once each. No compilers tested performed common subexpression elimination on these references. In general, such an optimization is inhibited by the possibility of side effects in the functions. Internal functions are more likely to be targets for common subexpression elimination than external functions (because the source code is present for the compiler to check for side effects), but none of the compilers tested performed such optimization, and the performance of internal and external functions is identical.
2. The three two-dimensional array expressions in the main loop are conformable, although no compilers tested fused them. The external references for both the allocation of the temporary arrays and the calls to the derivative functions inhibited the fusion. Even if the allocation/deallocation was hoisted out of the loop and the derivative functions inlined, fusion would still be inhibited because the individual statements within the array functions are not conformable with the result.

   The array references were also written to be conformable because Fortran 90 does not provide a natural way of subscripting the output of an array-valued function. Because the natural range of u and v differs, what I wanted to write was something of the form:

```
U(2:M-1,:,NEW) = U(2:M-1,:,OLD) &
  + D_DX(U(:,:,MID)) (2:M-1,:)
V(:,2:N-1,NEW) = V(:,2:N-1,OLD) &
  + D_DY(V(:,:,MID)) (:,2:N-1)
```

   where the second array section specification of the last term applies to the implicit temporary array holding the output of the d_dx function. It would be possible to simulate this using another array-valued function to provide the desired subset, but it is far more convenient to either use an explicit temporary array, or as in this case, to modify the main loop to be conformable to the shape of the array-valued function.
3. This coding has one potential advantage relative to explicitly managing the temporaries (Example 5b), and that is that the temporary arrays used to hold the output of the d_dx and d_dy subroutines are now under direct compiler control, and may therefore be more likely to be eliminated or overlapped in storage than if they had been specified explicitly (as in the Fortran 77 code). Eliminating the temporary arrays would decrease the execution time and memory traffic demands, while overlapping their storage (which would require some code rearrangement) would decrease the memory required to run the task. Optimally overlapping the storage requires careful analysis of the range over which each temporary array is needed, and a compiler is more capable of doing this tedious work optimally and correctly than a human programmer.

3.2 Example 5b—"Explicit Temps"

A second implementation of the code was produced using explicit temporary arrays. Four of the six temporary arrays are used twice per time step, while the other two are used only once. I chose to put all six into explicit temporaries (rather than just the four that are reused) because the two remaining array-valued function references would have still been enough to inhibit loop fusion of the three pairs of loops forming the computational kernel. With all of the array-valued function references removed, the VAST90 compiler is able to fuse the loops because there are no potential data dependencies. The Cray f90 compiler did not fuse the loops, but did (of course) vectorize them.

```
! Example 5b --- ''Explicit Temps''
REAL,DIMENSION(M,N,3)::     U,V,H
REAL,DIMENSION(M,N)::       DUDX,DUDY,DVDX,DVDY,DHDX,DHDY
REAL,DIMENSION(M,N)::       F
```

```
! --- Begin time marching loop ---
DO ISTEP=1,NSTEPS
    OLD = MOD(ISTEP+2,3)+1
    MID = MOD(ISTEP+3,3)+1
    NEW = MOD(ISTEP+1,3)+1
    ! ------ interior calculations ------ !
    DUDX = D_DX(U(:,:,MID))
    DUDY = D_DY(U(:,:,MID))
    DVDX = D_DX(V(:,:,MID))
    DVDY = D_DY(V(:,:,MID))
    DHDX = D_DX(H(:,:,MID))
    DHDY = D_DY(H(:,:,MID))
    U(:,:,NEW) = U(:,:,OLD)      - 2*DT* (                &
                 U(:,:,MID)*DUDX+V(:,:,MID)*DUDY          &
                   -F*V(:,:,MID)-G*DHDX )

    V(:,:,NEW) = V(:,:,OLD)      - 2*DT* (                &
                 U(:,:,MID)*DVDX+V(:,:,MID)*DVDY          &
                   +F*U(:,:,MID)-G*DHDY )

    H(:,:,NEW) = H(:,:,OLD)      - 2*DT* (                &
                    U(:,:,MID)*DHDX + V(:,:,MID)*DHDY     &
                   +H(:,:,MID)*(DUDX+DVDY) )
    ! ------ boundary calculations ------ !
    U(1,:,NEW) = 0 ; U(M,:,NEW) = 0
    V(:,1,NEW) = 0 ; H(:,1,NEW) = 0
    V(:,N,NEW) = 0 ; H(:,N,NEW) = 0
END DO
```

3.3 Example 5c—"Pointers"

A simplification of the presentation of the algorithm can be made by using Fortran 90 pointers, as shown below. Except for the use of pointers instead of array sections, this version is almost identical to Example 5a, and performs similarly as well. It is included here because of its remarkable clarity and readability rather than its novel optimization properties. The derivative functions are the same as for Example 5a.

```
! Example 5c --- ''Pointers''
REAL,DIMENSION(M,N,3), TARGET::      U_ARRAY,V_ARRAY,H_ARRAY
REAL,POINTER::       U_OLD(:,:),U(:,:),U_NEW(:,:)
REAL,POINTER::       V_OLD(:,:),V(:,:),V_NEW(:,:)
REAL,POINTER::       H_OLD(:,:),H(:,:),H_NEW(:,:)
REAL,DIMENSION(M,N),TARGET::         F_ARRAY
REAL,POINTER::       F(:,:), SWAP(:,:)

! associate the pointers with array sections
U_NEW => U_ARRAY(:,:,3) ; U => U_ARRAY(:,:,2) ; U_OLD => U_ARRAY(:,:,1)
V_NEW => V_ARRAY(:,:,3) ; V => V_ARRAY(:,:,2) ; V_OLD => V_ARRAY(:,:,1)
H_NEW => H_ARRAY(:,:,3) ; H => H_ARRAY(:,:,2) ; H_OLD => H_ARRAY(:,:,1)
F => F_ARRAY(:,:)
```

```
DO ISTEP=1,NSTEPS
    U_NEW = U_OLD - 2*DT*( U*D_DX(U) +V*D_DY(U)-F*V + G*D_DX(H) )
    V_NEW = V_OLD - 2*DT*( U*D_DX(V) +V*D_DY(V)-F*U + G*D_DY(H) )
    H_NEW = H_OLD - 2*DT*( U*D_DX(H) +V*D_DY(U)-H*(D_DX(U)+D_DY(V)) )

    SWAP => U_NEW ; U_NEW => U_OLD ; U_OLD => U ; U => SWAP
    SWAP => V_NEW ; V_NEW => V_OLD ; V_OLD => V ; V => SWAP
    SWAP => H_NEW ; H_NEW => H_OLD ; H_OLD => H ; H => SWAP
END DO
```

3.4 Example 5d—"Inline Fortran 77"

The final implementation is obtained by inlining the derivative functions manually, and manually fusing the inner loops to reduce the entire kernel to a single-loop pair. The code below is Fortran 77—very similar results are available by specifying the inlined operations in Fortran 90 array notation, provided that a compiler directive is given to notify the compiler that no data dependencies exist (i.e., MID.NE.NEW). This code is clearly less readable and maintainable than any of the previous examples, but its performance is considerably higher.

```
! Example 5d --- ''Inline Fortran 77''
      REAL U(M,N,3), V(M,N,3), H(M,N,3)
      REAL F(M)
!--- begin time marching loop ---
      DO ISTEP = 1, NSTEPS
         OLD = MOD(ISTEP + 2,3) + 1
         MID = MOD(ISTEP + 3,3) + 1
         NEW = MOD(ISTEP + 1,3) + 1
    DO J=2,N-1
    DO I=2,M-1
      U(I,J,NEW) = U(I,J,OLD) - TWODT*(
     $     C1X*U(I,J,MID)*(U(I+1,J,MID)-U(I-1,J,MID))
     $   + C1Y*V(I,J,MID)*(U(I,J+1,MID)-U(I,J-1,MID))
     $   - F(J)*V(I,J,MID) + C2X*(H(I+1,J,MID)-H(I-1,J,MID)))
      V(I,J,NEW) = V(I,J,OLD) - TWODT*(
     $     C1X*U(I,J,MID)*(V(I+1,J,MID)-V(I-1,J,MID))
     $   + C1Y*V(I,J,MID)*(V(I,J+1,MID)-V(I,J-1,MID))
     $   + F(J)*U(I,J,MID) + C2Y*(H(I,J+1,MID)-H(I,J-1,MID)))
      H(I,J,NEW) = H(I,J,OLD) - TWODT*(
     $            C1X*(U(I,J,MID)*(H(I+1,J,MID)-H(I-1,J,MID))
     $                +H(I,J,MID)*(U(I+1,J,MID)-U(I-1,J,MID)))
     $           +C1Y*(V(I,J,MID)*(H(I,J+1,MID)-H(I,J-1,MID))
     $                +H(I,J,MID)*(V(I,J+1,MID)-V(I,J-1,MID))))
    END DO
  END DO
! ------ boundary calculations ------ !
     DO I = 1, M
        U(1,I,NEW) = 0                    ! ETC...
        U(N,I,NEW) = 0                    ! ETC...
        V(I,1,NEW) = 0                    ! ETC...
        H(I,1,NEW) = 0                    ! ETC...
        V(I,N,NEW) = 0                    ! ETC...
        H(I,N,NEW) = 0                    ! ETC...
     END DO
  END DO
```

3.5 Example 5e—"Optimized Fortran 77"

Further improvements in the performance of the inlined version of the code can be obtained by manually unrolling the outer loop (on the Cray C90), or by manually unrolling the inner loop and then manually performing scalar replacement of array elements and common subexpression elimination (on the IBM and SGI). The performance results for the modified code (which is not shown here) are not presented as a "fair" comparison (because even mature Fortran 77 compilers cannot optimize Example 5d this well), but rather as a good indication of the peak performance of the hardware for the algorithm and as an ambitious target for compilers.

3.6 Timing Results

To explore the ways in which some of the optimization issues discussed above are dependent on the hardware characteristics, the code segments above were inserted into timing programs and run on three platforms with significantly different performance characteristics:

1. A vector supercomputer—the Cray C90, with Cray f90 (versions 1.0.2.0 and 1.0.2.7).
2. A microprocessor-based machine with a small cache (32 kB) and a good ratio of main memory bandwidth to cache bandwidth (90 MB/s and 160 MB/s, respectively)—the IBM RS/6000-320, with Pacific Sierra VAST90 (version 2.06G5) and the IBM XLF 2.3 Fortran 77 compiler.
3. A microprocessor-based machine with a large cache (4 MB) and a poor ratio of main memory bandwidth to cache bandwidth (150 MB/s and 1200 MB/s, respectively)—the Silicon Graphics Power Challenge, with Pacific Sierra VAST90 (version 2.06G5) and the SGI/MIPS Fortran 77 compiler (version 4.0.1).

(Note that the Silicon Graphics results were cross-compiled in 32-bit mode and do not take advantage of most of the 64-bit hardware on the Power Challenge. The relative timings as a function of code size and coding style are still valid, however, and show the effect of the cache size and memory bandwidth clearly.)

In each case, the number of time steps was adjusted so that the number of grid points times the number of time steps was held constant.

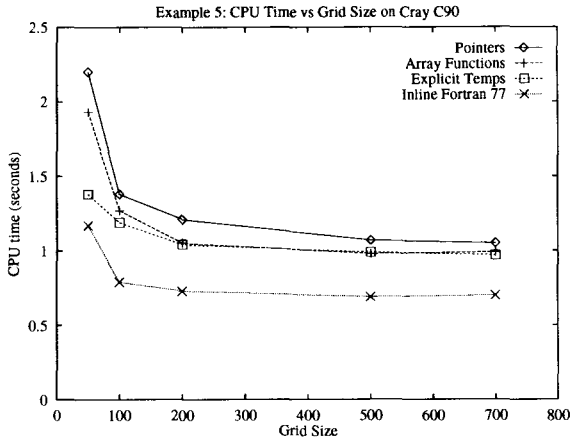

FIGURE 3 CPU time vs. problem size for four different codings of Example 5 on the Cray C90. The number of steps is adjusted so that the number of grid points times the number of time steps is constant.

The compiler choices are intended to be representative of the best of currently available systems, but I caution that this is merely a case study and not a systematic evaluation of either these two products or the whole range of Fortran 90 compilers. Some tests were performed with other Fortran 90 compilers, but the test coverage was not thorough enough to justify drawing conclusions or making comparisons at this time.

The total execution time as a function of coding style and grid size is presented in Figures 3–5 for the Cray C90, IBM RS/6000-320, and SGI Power

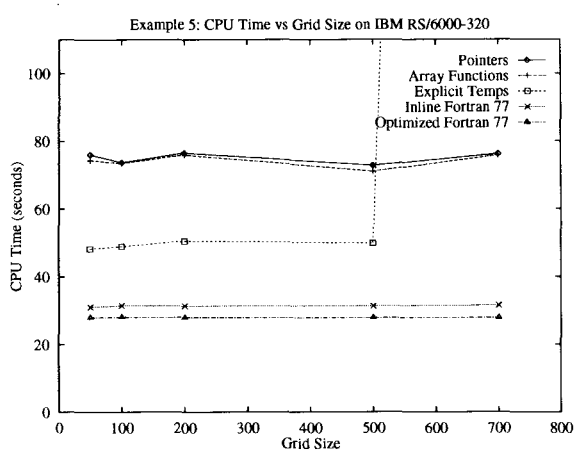

FIGURE 4 CPU time vs. problem size for four different codings of Example 5 on the IBM RS/6000-320. The number of steps is adjusted so that the number of grid points times the number of time steps is constant.

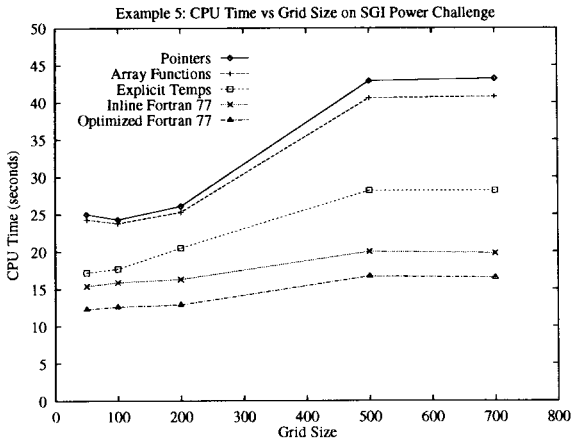

FIGURE 5 CPU time vs. problem size for four different codings of Example 5 on the SGI Power Challenge. The number of steps is adjusted so that the number of grid points times the number of time steps is constant.

Challenge, respectively. The contrasts between the performance trends nicely illustrate the issues involved:

1. On the Cray C90, the performance decreases significantly for the short vector lengths (recall that the number of time steps was adjusted to make the nominal total work constant), as expected, but we also see that the relative penalty for the cases with dynamic memory allocation ("pointer" and "array function") is significantly higher than for the statistically allocated cases. For large arrays, the difference between static and dynamic allocation disappears completely, and the ratio of worst-case to best-case time is only 1.5 : 1.
2. On the IBM RS/6000-320, the performance is almost perfectly uniform with problem size (because even the smallest case does not fit in cache). There is a significant penalty for the cases with dynamically allocated arrays, but this is due to the four extra subroutine calls and the lack of fusion in the inner loops, rather than to dynamic memory allocation overhead. The "pointer" and "array function" versions took 2.4 times as long to complete as the inline version, and about 1.5 times as long as the case with explicit temporaries. Note that the largest case with explicit temporaries could not be run on my 64-MB workstation—this is a good example of the cost of having too many temporary arrays.
3. The SGI Power Challenge shows a clear transition in performance N = 200 (cached) and N = 500 (uncached). As on the IBM, and unlike the Cray, the penalty suffered (by the dynamically allocating versions) for the extra function calls and the failure to fuse the inner loops is relatively large.

The pattern of the performance ratios on the SGI is a bit more clear in the CPU time ratios shown in Figure 6. There is some evidence of extra overhead costs of dynamic memory allocation for the smallest sizes, and it is clear that at the largest sizes, the dynamically allocated versions pay a larger relative penalty for their increased data movement.

In all cases, manual optimization provided improvements in the 5–10% range, indicating that the compilers did a competent job at optimization of the inlined version. On both the Cray and IBM, the manually inlined Fortran 90 version performed about 5–10% slower than the manually inlined Fortran 77 version. Such small differences in performance are not of particular concern or interest here.

3.7 Optimization of Operations and Memory Traffic

More detailed measurements of the performance characteristics include counts of the floating-point operations and memory operations of each of the example codes. On the Cray C90, such operation

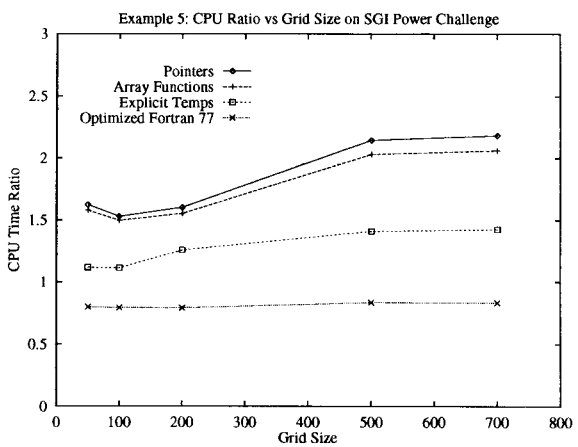

FIGURE 6 Ratio of the CPU time of each case to that of the "inline Fortran 77" version on the SGI Power Challenge. The number of steps is adjusted so that the number of grid points times the number of time steps is constant.

Table 1. Performance Data From the Cray C90, Showing the Number of Arithmetic and Memory Operations per Grid Point per Time Step of the Nonlinear Shallow Water Model

Version	Floating-Point		Memory		Loop Balance
	Additions per Point	Multiplies per Point	Loads per Point	Stores per Point	
Pointer	22	24	47	16	0.73
Array Function	22	24	44	13	0.80
Explicit Temps	18	20	42	15	0.66
Inline	20	21	23	3	1.56

NOTE: The "Loop Balance" is defined as the ratio of arithmetic operations to memory operations.

counts are provided by the hardware performance monitor, while on the IBM RS/6000, they were obtained by visual inspection of the generated object code.

For the Cray C90, Table 1 shows the measurements, expressed in operations per grid point per time step. The results demonstrate the slight increase in work required for the cases using array-valued functions. This has two sources: (1) the scalar multiplication inside the array-valued functions cannot be combined with the scalar multiplications in the array statement referencing the array-valued function and (2) the derivative functions are called four extra times relative to the inlined or explicit temporary code. The increase in memory traffic is more interesting, going from 26 memory references per grid point per time step to 63 references per grid point per time step in the pointer-based case. This increase is due to the use of many temporary arrays (either explicit or dynamically allocated) which must typically be stored to main memory (because there are only eight vector registers on the machine).

For the IBM RS/6000, the results are presented in Table 2, again expressed in operations per grid point per time step. The specific timing results are presented for the 500 × 500 grid. The last column presents the "lost" cycle count, which is a combination of work done outside the inner loops and cache miss waiting time. The amount of "lost" time triples from the inlined to noninlined cases—so much that the entire inlined calculation could have been performed during the "lost" cycles of the versions that used the array-valued functions. The "optimized" version made use of manual common subexpression elimination and array element scalar replacement to eliminate three redundant floating-point operations and five redundant loads that the XLF compiler was not able to optimize away.

The arithmetic counts are identical to those on the Cray, but the memory traffic differs significantly in one case. With explicit temporaries, the Cray required 15 extra loads and 6 extra stores per grid point per time step. This extra traffic was generated because there were not enough vector registers to hold all the data items needed to fully optimize the fused inner loop.

3.8 Memory Usage

Finally, we consider the increase in memory usage of the Fortran 90 versions of the code. The maximum amount of memory used by a job is reported

Table 2. Performance Data From the IBM RS/6000-320 Showing the Number of Arithmetic and Memory Operations per Grid Point per Time Step of the Nonlinear Shallow-Water Model

Version	Floating-Point Add/Mul per Point	Memory Loads/Stores per Point	Loop Balance	Estimated CPU Cycles per Point	Observed CPU Cycles per Point	"Lost" CPU Cycles per Point
Pointer	22/24	42/13	0.84	71	146	75
Array Functions	22/24	42/13	0.84	66	142	76
Explicit Temps	18/20	27/9	1.06	45	100	55
Inline	20/21	23/3	1.58	37	63	26
Optimized	18/18	18/3	1.71	31	56	25

NOTE: The "Loop Balance" is defined as the ratio of arithmetic operations to memory operations. The "Estimated" cycle counts are from the XLF compiler, while the "Observed" cycle counts are from the run-times. The "Lost" cycle count is the difference.

Table 3. Maximum Memory (MB = 2^{20} bytes) Used by the Four Codings of the Nonlinear Shallow-Water Equations From Section 3

	Cray cf77/f90	IBM VAST90
Array Functions	28.0	27.1
Explicit Temps	33.8	30.6
Pointers	30.0	27.0
Inline	18.5	17.3

by the "ja" utility on the Cray, and by the tcsh "time" utility on the IBM RS/6000. The results are presented in Table 3. For both the Cray f90 compiler and the VAST90 compiler, the array-based version of the code took almost 50% more memory than the inlined version. As discussed previously, the "Explicit Temps" version declared more temporary arrays than it really needed, so no dynamically declared temporary arrays would be generated in the inner loops and thus inhibit loop fusion.

In many cases the increase in the data space required by a program will be of no consequence, but when it becomes a problem, it is a large one. On virtual memory machines, exceeding the available physical memory can cause extremely poor performance due to thrashing, while on nonvirtual memory machines (such as the Cray), exceeding the available physical memory causes the job to abort.

4 DISCUSSION

4.1 Balance

From these performance data, we can calculate the machine balance of a computer system as the ratio of the sustainable FP operation rate (in operations per cycle) to the sustainable memory transfer rate (in words per cycle) [4, 5].

$$\beta_M = \frac{FP_{rate}}{MEMORY_{rate}} \tag{9}$$

In contrast to [4, 5], I added a rough consideration of the effects of memory hierarchy in the definition of the memory access rate, by assuming that memory references are uncached, but contiguous (unit stride). Examples of observed machine balance from current high-performance computers are presented in Table 4 (from [11]). Examples of machines with numerically lower machine balance are less common today, but include most machines with software floating-point implementations, such as the original Connection Machine CM-1 and those CM-2s without floating-point hardware.

A loop balance may be defined in a completely analogous way for a particular calculation.

$$\beta_L \equiv \frac{\#\text{ of FP operations}}{\#\text{ of Memory Accesses}} \tag{10}$$

This quantity is also sometimes referred to as the "computational density" or "compute intensity" [6–8].

When the loop balance is greater than the machine balance, the loop is said to be compute bound. On the other hand, when the loop balance is less than the machine balance, the loop is said to be memory bound. When the loop balance and machine balance are equal, the loop is said to be balanced.

Finally, we can define the ratio of the machine balance to the loop balance:

$$\delta \equiv \frac{\beta_M}{\beta_L} \approx \frac{MEMORY_{time}}{FP_{time}} \tag{11}$$

so that $\delta > 1$ for memory-bound calculations on a given machine and $\delta < 1$ for CPU-bound calculations on a given machine.

Thus, from Table 5, we see that the inline case is compute bound on the Cray C90, the case with

Table 4. Sustainable FP Operation Rate, Memory Transfer Rate, and Machine Balance of Five Modern High-Performance Computers, Including Those Discussed in This Study

Machine	FP ops/clock	Words/Clock	Machine Balance
Cray C90	3.7	5.00	0.74
Cray J90	1.8	1.31	1.37
IBM RS/6000-990	4.0	1.25	3.20
IBM RS/6000-320	2.0	0.38	5.26
SGI Power Challenge	4.0	0.225	17.78

Table 5. Loop Balance (β_L), Machine Balance (β_M), and Their Ratio ($\delta = \beta_M/\beta_L$) for the Two Systems Tested Here

	IBM RS/6000-320			Cray C90		
Version	β_L	β_M	δ	β_L	β_M	δ
Pointer	0.84	5.26	6.26	0.73	0.74	0.99
Array Functions	0.84	5.26	6.26	0.80	0.74	0.92
Explicit Temps	1.06	5.26	4.96	0.66	0.74	1.12
Inline	1.58	5.26	3.33	1.56	0.74	0.47

NOTE: Calculations are memory bound if $\delta > 1$ and CPU-bound if $\delta < 1$.

explicit temporaries is slightly memory bound, and the other two cases are balanced. All the cases are memory bound on the IBM RS/6000-320, although the inlined code is closest to being balanced.

4.2. Inlining

A feature of the examples above that accounts for a large fraction of the performance reduction is that none of the compilers tested are able to inline array-valued functions. The internal function definitions used to define the derivative operators are converted to external functions by the compiler, and called as standard subroutines. These subroutine calls inhibit optimizations such as loop fusion. Inlining is expected to be a feature of future versions of these compilers, and this should help the performance somewhat—not by reducing the subroutine call overhead, but by allowing more optimization between consecutive loop sets.

For two reasons, inlining can be expected to be more effective on hierarchical memory machines than on vector machines. First, inlining (typically) allows the reduction of loads and stores as much as it allows the reduction of arithmetic operations. On hierarchical memory machines, the loads and stores have a much higher relative cost than on a vector machine. Second, vector machines typically have a very small number of vector registers, and therefore very little room to store common subexpressions. Complicated expressions quickly spill their temporaries back to main memory, as has been seen in the examples above.

4.3 "Active" Loop Fusion

Because array-valued expressions must be conformable to be fused, it would be useful to develop a class of optimizations that produce conformable sequences of statements from certain classes of nonconformable sequences of statements.

It is likely that such optimizations will be based on two parts: a statement permuter and a multidimensional "peeler" (or "anti-peeler," as in Example 4). Current loop fusion algorithms are entirely passive—if the two loops are adjacent, conformable, and not data dependent, then they are fused. "Active" loop fusion will relax the first requirement by permuting the order of the statements (subject to the limits of data independence) to make potentially conformable statements adjacent (as was done manually in Example 3). When these adjacent loops are "almost" conformable, a multidimensional generalization of loop peeling can be used to attempt to find the greatest common subarray size to be chosen for the fusion. The remainders (typically edges) will be handled in a separate portion of code. These "clean-up" code sections may also be moved forward or backward in the instruction stream (subject to the limits of data independence) to enable the optimizer to try more loop fusions.

4.4 Data-Dependency Directives

To help the compiler with data-dependency analysis, a standard set of compiler directives should be adopted by the industry. These directives should include simple assertions, as in

```
!PRAGMA ASSERT  (NEW.NE.OLD)
U(2:M-1,  NEW)  = U(3:M,OLD)-U(1:M-2,OLD)
!PRAGMA END ASSERT  (NEW.NE.OLD)
```

or

```
!PRAGMA ASSERT NO SIDE EFFECTS
! applies to next subroutine call
```

as well as more sophisticated block structures, which would indicate to the compiler that an entire range of array statements contains no data dependencies. An example may look like

```
!PRAGMA BEGIN DATA-INDEPENDENT BLOCK
U_NEW = U_OLD + ...
V_NEW = V_OLD + ...
H_NEW = H_OLD + ...
!PRAGMA END DATA-INDEPENDENT BLOCK
```

This last example would assist the compiler in determining the scope in which to search for common array-valued expressions and reused array sections. It would also be useful for specifying parallelism, although that is not its primary intent.

Specific assertions (such as NEW.NE.OLD) are preferred over generalized assertions, which typically turn off all dependence checking in a statement, because they are less likely to be inadvertently incorrect. An industry-wide standard should be agreed upon because of the difficulty of correctly maintaining several independent sets of compiler directives in one's code. In addition to the obvious difficulty in trying to remember each major vendor's directive syntax, errors in directives are likely to be ignored rather than reported, resulting in performance degradation that is tedious to track down.

Unfortunately, simple assertions may not be adequate for all applications. Some might require very extensive logical constructs and others might require access to user-defined functions. It is not reasonable to allow the latter, because these could then be part of the executable program through their side effects, rather than being simple comments.

As mentioned previously, users can make use of the Fortran requirement that modified arguments to subroutines/functions to be free of aliasing to provide extra information to the compiler. Internal subroutines are particularly useful in this regard, because they inherit all the variables from the parent subroutine. An internal subroutine could have been used in Example 5 to update each of the arrays, implicitly declaring that each LHS array variable was not aliased with any of the RHS variables used in the update. If the code is inlined, then no performance penalty should accrue. If the code is not inlined, then the compiler may have difficulty hoisting temporary array allocation or minimizing memory traffic, and these can cause performance reductions for small and large programs, respectively.

4.5 Memory Management Directives

An additional set of directives is required for helping the compiler choose a strategy in the trade-off of memory vs. number of calculations. Extra temporary arrays to hold reused array-valued expressions may make the code run faster in many circumstances, but may make it run slower (or not at all) in cases with limited cache or physical memory. These issues are relatively new with Fortran 90 (though one could raise the same issues in vectorized Fortran 77 code), because common subexpression elimination typically creates scalar temporaries in Fortran 77, but will create array temporaries in Fortran 90 if the compiler cannot prove the absence of data dependencies.

It can be quite difficult for even the user to know which approach will be best for a given code, so it is unlikely that the compiler will always make the right decision. Unfortunately, compilers (except for source-to-source translators) do not provide enough information for the user to even know what the compiler is doing, which makes it considerably more difficult to provide useful hints.

A difficulty in optimizing a Fortran 90 code is that it is often very difficult for the user to determine exactly what the compiler has done with the code. The Pacific Sierra VAST90 processor is helpful in that regard, because it emits Fortran 77 code as an intermediate representation. In other cases, one must rely on compiler-generated listings (if available) or direct examination of the machine code in order to discover what optimizations the compiler has performed.

5 CONCLUSIONS

From these examples, it is possible to use Fortran 90 array and pointer notation to significantly simplify the presentation of the shallow-water equation algorithm, although in several cases at some cost in performance and/or memory usage. The issues covered in this analysis will also apply to Fortran 90 implementations of many algorithms for the solution of partial differential equations, and some algorithms for sparse linear algebra. Although these constitute an important class of scientific programs, it would be inappropriate to generalize too broadly from these results.

Given that caveat, for this class of algorithms, the increased ease of coding, reliability, and modularity of the Fortran 90 versions will often outweigh the performance concerns. On the other hand, for computational jobs that require the highest performance on either hierarchical memory or vector computers, manual optimizations that decrease

the clarity of the code may be required. For the examples explored here, suitably optimized Fortran 90 array notation (in conjunction with the best Fortran 90 compilers) was capable of generating approximately the same performance as well-constructed Fortran 77 loops. These, in turn, provided about 90% of the performance of hand-optimized Fortran 77 code.

For the most part, the penalties of poorly optimized Fortran 90 array notation are not large (typically factors of 1.5 to 2.5 in the examples considered here). However, there are specific cases where the penalties can be much more severe:

1. Temporary arrays are allocated and deallocated too frequently.
2. Extra temporary arrays cause a code (or a significant portion of a code) to exceed the machine's cache size.
3. Extra temporary arrays cause the code's working set to exceed the machine's available physical memory.

In a multiuser batch environment, similar slowdowns in effective throughput could result if (because of extra temporaries) a job had to be moved to a lower-priority queue with a larger working set allocation.

We have seen several examples for which array operations coded in Fortran 90 suffer from significant increases in memory traffic requirements. If maximum performance is the primary concern, this is exactly the wrong direction to go, given the current trend toward greater and greater CPU power per unit of memory bandwidth. The performance results presented above are likely to be worse when considering workload performance on shared memory machines, as the increased demand for memory traffic will result in increased memory bank contention.

It is difficult, at this rather early stage in the development of Fortran 90 compilers, to determine what fraction of the speed degradation of each of these examples is due to the language semantics introducing insoluble data-dependency questions and how much is due to immaturity of the compilers. There is no question that compiler immaturity is a significant issue in the results presented here (e.g., the compilers frequently made poor decisions regarding temporary array allocation), but it still appears that there are pieces of the Fortran 90 array notation optimization process which are intrinsically more difficult than for Fortran 77 loops.

ACKNOWLEDGMENTS

I thank John Prentice and Reg Clemens of Quetzal Computational Associates, Preston Briggs of Tera Computer, and Jon Steidel and Bill Homer of Cray Research for helpful comments on earlier drafts of the manuscript, and two anonymous reviewers for extremely thorough critiques of an earlier draft.

REFERENCES

[1] R. Allen and K. Kennedy, "Automatic translation of Fortran programs to vector form," *ACM Trans. Programming Languages Systems*, vol 9, pp. 491–542, Oct. 1987.

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surveys*, vol. 26, p. 345ff, 1994.

[3] M. Byler, J. R. B. Davies, C. Huson, B. Leasure, and M. Wolfe, "Multiple version loops," in *Proceedings of the 1987 International Conference on Parallel Processing*, Aug. 1987, pp. 312–318.

[4] D. Callahan, J. Cocke, and K. Kennedy, "Estimating interlock and improving balance for pipelined architectures," in *Proceedings of the 1987 International Conference on Parallel Processing*, Aug. 1987, pp. 295–304.

[5] D. Callahan, J. Cocke, and K. Kennedy, "Estimating interlock and improving balance for pipelined architectures," *J. Parallel Distrib. Comput.*, vol. 5, pp. 334–358, 1988.

[6] B. R. Carlile, "Algorithms and design: The Cray APP shared-memory system," in *COMPCON '93*, Feb. 1993, pp. 312–320.

[7] R. W. Hockney and C. R. Jesshope, *Parallel Computers*. Philadelphia: Adam Hilger, 1981, pp. 106–108.

[8] R. Hockney, "r_∞, $n_{1/2}$, $s_{1/2}$ measurements on the 2 CPU CRAY X/MP," *Parallel Comput.*, vol. 2, pp. 1–14, 1985.

[9] J. D. McCalpin, "A comment on 'Performance benchmark results for selected Fortran 90 compilers' by Prentice and Ameko," *Fortran J.*, vol. 7, p. 16ff, May/June 1995.

[10] J. D. McCalpin, "Rossby wave generation by poleward propagating Kelvin waves: The low-frequency, quasigeostrophic approximation," *J. Phys. Oceanogr.*, vol. 25, pp. 1415–1425, 1995.

[11] J. D. McCalpin, "A survey of memory bandwidth and machine balance in current high performance computers," A continually updated technical report, available at http://perelandra.cms.udel.edu/~mccalpin/hpc/balance., 1995.

[12] J. K. Prentice and A. K. Ameko, "Performance benchmarks for selected Fortran 90 compilers," *Fortran J.*, vol. 6, pp. 2–10, 1994.

[13] C. Schimmel, *UNIX Systems for Modern Architectures*. Reading, MA: Addison-Wesley, 1994.