The Implementation and Evaluation of Fusion and Contraction in Array Languages*

E Christopher Lewis Calvin Lin† Lawrence Snyder

University of Washington, Seattle, WA 98195-2350 USA
†University of Texas, Austin, TX 78712 USA
{echris,snyder}@cs.washington.edu, lin@cs.utexas.edu

Abstract

Array languages such as Fortran 90, HPF and ZPL have many benefits in simplifying array-based computations and expressing data parallelism. However, they can suffer large performance penalties because they introduce intermediate arrays—both at the source level and during the compilation process—which increase memory usage and pollute the cache. Most compilers address this problem by simply scalarizing the array language and relying on a scalar language compiler to perform loop fusion and array contraction. We instead show that there are advantages to performing a form of loop fusion and array contraction *at the array level.* This paper describes this approach and explains its advantages. Experimental results show that our scheme typically yields runtime improvements of greater than 20% and sometimes up to 400%. In addition, it yields superior memory use when compared against commercial compilers and exhibits comparable memory use when compared with scalar languages. We also explore the interaction between these transformations and communication optimizations.

1 Introduction

Array languages such as Fortran 90 (F90) [1], High Performance Fortran (HPF) [13] and ZPL [24] have become important vehicles for expressing data parallelism. Though they simplify the specification of array-based calculations, they also present a potential problem: Large temporary arrays may need to be introduced, either by the programmer or by the compiler. For example, the F90 statements in Figure 1(a) use temporary array R to cache a computation, while the Fortran 77 equivalent in Figure 1(b) uses only the scalar variable s, which can be viewed as a *contracted* form of the full array R. Similarly, there are cases where an array language compiler will insert temporary arrays to preserve array language semantics. In both cases, these array temporaries increase memory usage, degrade performance by polluting the cache, and therefore impede the acceptance of array languages, despite their other advantages.

There are two ways to solve this problem. The first is to scalarize the array language source (*i.e.*, produce scalar loop nests for each array statement) and rely on a scalar language compiler to remove the array temporaries using its existing *scalar level* optimizations. Specifically, the scalar compiler must fuse loops to enable contraction, as shown in Figure 1(b). The second approach is to optimize at the *array level* prior to scalarization (*i.e.*, perform analyses and transformations on array statements directly). Since fusion and contraction are mature and well understood transformations, the first approach appears a natural choice because it simplifies the array language compilation process and leverages existing compiler technology. However, we believe that the first approach is inferior for several reasons. First, the reality is that scalar languages do not require programmers to introduce large temporary arrays, so scalar language compilers do not bother to perform costly transformations that will not benefit typical human generated code. Second, removing array temporaries at the scalar level solves the problem at a greater conceptual distance from the source of the problem and at a greater cost. Most importantly, it is impractical to implement an integrated optimization strategy when some optimizations (*e.g.*, communication pipelining) are performed at the array level while others (*e.g.*, array contraction) are subsequently performed at the scalar level.

Our work supports earlier claims that there are performance benefits to performing analyses and transformations at the array level [6, 21]. In particular, this paper makes the following contributions. We explain how fusion and contraction can be performed at the array level. We refer to the former as *statement fusion* because array statements, not loops, are the fused entities. We provide empirical evidence that our approach is superior to those used by several commercial compilers. We measure the benefits of various array level fusion and contraction strategies, both in terms of execution time and memory usage, and we find that the common practice of contracting only compiler introduced arrays is insufficient. In addition, we show that our array level approach produces code that is comparable to that of hand-written scalar programs. Finally, we show that performance suffers when compilers do not use an integrated strategy for optimizing communication and performing fusion.

This paper is organized as follows. Sections 2 and 3 define the representations we use and the problem we solve. Section 4 describes our solution to the problem. Section 5 evaluates the implementation of statement fusion and array contraction in the ZPL compiler, and the final two sections present related work and give conclusions.

*This research was supported in part by DARPA Grant E30602-97-1-0152 and NSF Grant CCR-9710284.

```
R(i,:)=AA(i,:)*D(i-1,:)
D(i,:)=1.0/(DD(i,:)-AA(i-1,:)*R(i,:)
Rx(i,:)=Rx(i,:)-Rx(i-1,:)*R(i,:)
Ry(i,:)=Ry(i,:)-Ry(i-1,:)*R(i,:)
```

(a)

```
do j=1,n
   s = AA(i,j)*D(i-1,j)
   D(i,j)=1/(DD(i,j)-AA(i-1,j)*s
   Rx(i,j)=Rx(i,j)-Rx(i-1,j)*s
   Ry(i,j)=Ry(i,j)-Ry(i-1,j)*s
enddo
```

(b)

Figure 1: Illustration of unnecessary array allocation (R) in an array language using a code fragment from the tridiagonal systems solver component of the SPEC CFP95 Tomcatv benchmark.

2 Representations

This section describes our array statement normal form and array level dependence representation. The representation we describe will be used in defining the fusion for contraction problem and in implementing its solution.

2.1 Normalized Array Statements

We define a *normalized array statement* to be an element-wise array operation that has the following properties: (i) the same array (or aliasing arrays) may not be both read and written, (ii) the statement contains arrays of a common rank, and (iii) the extent of the array statement's computation is defined by an index set, called a *region*, and all array references are specified as constant offsets from this index set. The final property implies that for each array reference, the elements of its subscript are *separable* (*i.e.*, an index variable may appear in only a single element of a subscript) and a particular index variable appears in the same position of all subscripts. A normalized array statement has the following form.

$$[\mathsf{R}]\ f(\mathsf{A}_1@\mathsf{d}_1,\ \mathsf{A}_2@\mathsf{d}_2,\ldots,\ \mathsf{A}_s@\mathsf{d}_s)$$

The indices of array A_i involved in the computation are those of the region $\mathsf{R}=[1..\mathsf{n}_1,1..\mathsf{n}_2,\ldots,1..\mathsf{n}_r]$ offset by the integer r-tuple $\mathsf{d}_i=(\mathsf{d}_{i_1},\ldots,\mathsf{d}_{i_r})$, *i.e.*, $[1+\mathsf{d}_{i_1}..\mathsf{n}_1+\mathsf{d}_{i_1},\ldots,1+\mathsf{d}_{i_r}..\mathsf{n}_r+\mathsf{d}_{i_r}]$. Figures 2(a) and (b) contain F90 array statements and their normal form equivalents (note that $\mathsf{A} \equiv \mathsf{A}@0$, where 0 is the null or zero vector). Though only normalized statements can participate in fusion and contraction, unnormalized statements do not prevent independent normalized statements from being optimized.

This normal form is an appropriate representation for array statements because the data volume of each term in a single array statement is the same (*i.e.*, they are conformable). Conformability permits most F90 and ZPL array statements to be normalized, either directly or by the introduction of compiler temporaries that are often later contracted. Furthermore, the normal form serves as an effective internal representation when compiling for parallel machines because it makes the alignment of arrays explicit. All array references are perfectly aligned except for vector offsets, so normalized statements will compile to highly efficient parallel code [7]. Compiler generated communication primitives need not be normalized because they are not candidates for fusion or contraction. Not coincidentally, the normal form closely resembles the core of the ZPL source language.

```
1  A(1:m,1:n) = B(0:m-1,1:n)
2  C(1:m,1:n) = A(1:m,0:n-1)
3  B(1:m,1:n) = A(0:m-1,2:n+1)
```

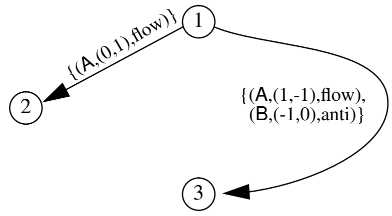
(a)

```
1  [1..m,1..n] A := B@(-1,0);
2  [1..m,1..n] C := A@(0,-1);
3  [1..m,1..n] B := A@(-1,1);
```

(b)

```
do j = n,1,-1
   do i = m,1,-1
      A(i,j) = B(i-1,j)
      B(i,j) = A(i-1,j+1)
   enddo
enddo
do i = 1,m
   do j = 1,n
      C(i,j) = A(i,j-1)
   enddo
enddo
```

(c)



(d)

Figure 2: Four different representations of the same array computation: (a) Fortran 90, (b) normalized array statements, (c) Fortran 77, and (d) array statement dependence graph.

2.2 The Array Statement Dependence Graph

In this section, we review the concept of data dependence, and we modify existing mechanisms to represent dependences between normalized array statements. Data dependences [26] represent ordering constraints on statements in a program. A *flow* or *true dependence* requires that a variable assignment precede a read to the same variable, and an *anti-dependence* requires the reverse. An *output dependence* requires that one assignment to a variable precede another assignment to the same variable. Transformations that reorder dependent statements (*i.e.*, move the dependence target before its source) are illegal because they violate the dependence and do not preserve correctness.

Data dependence is also used to represent ordering constraints on iterations in the iteration space of a loop nest. The iteration space associated with a loop nest has a dimension for each loop in the nest. Loop transformations such as loop interchange or loop reversal are only legal if they preserve the data dependences in the iteration space. *Distance vectors* serve as a static analysis tool to represent data dependences concisely in an iteration space.

Definition 1 *A* distance vector *is an integer n-tuple, $d = (d_1, d_2, \ldots, d_n)$, representing a dependence between the iterations of a rank n iteration space, where the source of the dependence precedes the target by d_i iterations in loop i (1 is the outermost), for $1 \le i \le n$. Note that a negative or zero value implies that the target precedes the source or that they are in the same iteration, respectively.*

A distance vector is *lexicographically nonnegative* if it is a null vector or if its leftmost non-zero element is positive. A lexicographically nonnegative distance vector is said to be *legal* because the dependence source precedes the target in the loop that *carries* the dependence. If a distance vector is not lexicographically nonnegative, then the target of the dependence precedes the source in the loop that carries the dependence, which is clearly illegal.

Distance vectors are inappropriate for use in array level compilation because they are derived from loop nests, which are not created until after our transformations have been performed. As a result, we introduce a variant of the distance vector, called the *unconstrained distance vector*, to represent array level data dependences between normalized array statements.

Definition 2 *An* unconstrained distance vector *is an integer n-tuple, $u = (u_1, u_2, \ldots, u_n)$, representing a dependence between two normalized n-dimensional array statements, where the source of the dependence precedes the target by u_i iterations of the loop that iterates over dimension i (if both statements appear in the same loop nest).*

Unconstrained distance vectors are constructed by subtracting the dependence's target offset vector from its source offset. For example, the unconstrained distance vectors that arise from the dependences in the code in Figure 2(b) are $(0,0)-(0,-1) = (0,1)$ and $(0,0)-(-1,1) = (1,-1)$ for array A and $(-1,0)-(0,0) = (-1,0)$ for array B. The lexicographical nonnegativity of an unconstrained distance vector has no bearing on the legality of the dependence it represents.

Because scalarization of a normalized statement generates a single loop to iterate over the same dimension of all arrays in its body, we can characterize dependences by dimensions of the array rather than dimensions of the iteration space. Thus u_i is the distance of the dependence along array dimension i. Unconstrained distance vectors are more abstract than traditional (constrained) distance vectors because they separate loop structure from dependence representation. Though unconstrained distance vectors are not fully general, they can represent any dependence that appears in our normal form.

We represent code using the *array statement dependence graph.*

Definition 3 *An* array statement dependence graph *(ASDG), $G = (V, E)$, is a labeled, acyclic, directed graph, where vertices, v_i, represent statements, edges represent data dependences between statements, and each edge, $(v_1, v_2) \in E$, is labeled, $l(v_1, v_2)$, with a set of (variable name, unconstrained distance vector, dependence type $\in$ {flow, anti, output}) tuples.*

An ASDG is guaranteed to not contain cycles because it represents a single basic block at the array statement level. An edge from v_1 to v_2, $(v_1, v_2) \in E$, in an ASDG indicates that the target statement v_2 is dependent on the source statement v_1. The label on each edge in the ASDG describes the dependences the edge represents by naming the variables that induce the dependences and the associated unconstrained distance vectors and dependence types. Figure 2(d) contains the ASDG that corresponds to the normalized array statements in 2(b).

If after scalarization, the source and target of a dependence appear in the single loop nest, a conventional (constrained) distance vector may be constructed from an unconstrained one given a description of the loop nest structure.

Definition 4 *A* loop structure vector *is an integer n-tuple, $p = (p_1, p_2, \ldots, p_n)$, that describes the dimension and direction of each loop in an n-deep loop nest. Loop i (1 is the outermost loop in the loop nest) iterates over dimension $|p_i|$ in the direction of the sign of p_i, positive denoting increasing.*

A loop structure vector is a permutation of $(\pm 1, \pm 2, \ldots, \pm n)$. The loop structure vector that describes the loop nests in Figure 2(c) are $(-2,-1)$ and $(1,2)$. In the first nest, the outer loop iterates over the second dimension and the inner loop iterates over the first dimension, both in a decreasing direction.

A constrained distance vector, $d = (d_1, d_2, \ldots, d_n)$, is constructed from an unconstrained one, u, and a loop structure vector, p, by letting $d_i = \frac{p_i}{|p_i|} u_{|p_i|}$, for $1 \le i \le n$. Consider array statements 1 and 3 in Figure 2(b). If $p = (-2,-1)$, the unconstrained distance vectors $(-1,0)$ and $(1,-1)$ become $(0,1)$ and $(1,-1)$, respectively, when constrained. The constrained distance vectors are lexicographically nonnegative, so the dependences of the code in Figure 2(b) are preserved by the first loop nest in 2(c) resulting from loop structure vector p. There are no constraints on the structure of the second loop nest because it does not contain statements that depend on each other.

A *fusion partition* describes a particular fusing of the statements in an ASDG.[1]

Definition 5 *A* fusion partition*, $P = (P_1, P_2, \ldots, P_l)$, of an ASDG, $G = (V, E)$, is a partitioning of the nodes of G into l disjoint sets, $P_1, P_2, \ldots, P_l$, called* fusible clusters *such that the following conditions hold: (i) all statements in a single cluster operate under the same region, (ii) all unconstrained distance vectors on intra-fusible-cluster flow dependences are null vectors (i.e., $\forall P_i$ and $v_1, v_2 \in P_i$, if $(x, u, flow) \in l(v_1, v_2)$ then u is a null vector), (iii) there are no inter-fusible-cluster cycles, and (iv) a loop structure vector exists for each fusible cluster that preserves all intra-fusible-cluster dependences.*

Upon scalarization, all the statements in a fusible cluster are implemented with a single loop nest. The statements in each loop nest and the loop nests themselves are ordered by a topological sort using intra- and inter-fusible cluster dependence edges, respectively. The first condition above ensures that all the statements in a single cluster have the same (*i.e.*, conformable) loop bounds. The second condition ensures that a loop carried flow dependence will not inhibit parallelism. The final two conditions ensure that inter- and intra-fusible-cluster dependences are preserved, respectively. An algorithm to decide the final condition is described in detail in Section 4.2. The *trivial fusion partition* of an ASDG is one in which there is exactly one statement in each fusible cluster.

Given a particular fusion partition we can decide for what arrays contraction has been enabled.

Definition 6 *Given a fusion partition, $P = (P_1, P_2, \ldots, P_l)$, of an ASDG, $G = (V, E)$, an array, x, is* contractible *if the following conditions hold: (i) the source and target of all dependences due to x appear in the same fusible cluster (i.e., $\forall (v_1, v_2) \in E$, if $(x, u, t) \in l(v_1, v_2)$, then $v_1 \in P_i$ and $v_2 \in P_i$ for some i, $1 \le i \le l$), and (ii) the unconstrained distance vectors of all data dependences due to x are null vectors (i.e., $\forall (v_1, v_2) \in E$, if $(x, u, t) \in l(v_1, v_2)$, then u is a null vector).*

These conditions ensure that all references to x will appear in a single loop nest upon scalarization, and there will be no loop carried dependences due to x. The latter condition may be relaxed when the dependence is along a dimension of the array that is not distributed [4], but here we assume that all dimensions are distributed.

[1] This terminology is borrowed from Gao *et al.* [12], who considered a similar problem. See Section 6.

3 Problem

There are two reasons to perform statement fusion: to enable the elimination of arrays by contraction and to improve utilization of the data cache by exploiting inter-statement reuse. For the first goal, we seek a fusion partition, P, for an ASDG, G, that enables the maximum elimination of array element references by contraction. The number of array element references eliminated by the contraction of array x (called *reference weight*, $w(x, G)$) is a function of the number of times it is referenced at the array level and the region sizes over which these references occur. We call the sum of the reference weights of all contracted arrays the *contraction benefit* of a fusion partition. For the second goal, we seek a fusion partition that maximizes the number of arrays without inter-fusible-cluster dependences. The intuition is that while intra-cluster dependences are potential sources of cache reuse, we must be careful not to pollute the cache with the increased references that come with excessive fusion. When all references to an array appear in a single loop nest, all other loop nests are spared the cache burden of references to the array. Both problems are provably NP-complete, so we present approximate solutions in the next section.

4 Solution

This section presents algorithms for performing statement fusion to enable contraction and exploit locality. Because eliminating entire arrays conserves memory and can result in enormous performance improvements, we perform fusion for contraction first. We also describe the details of scalarization.

4.1 Statement Fusion

Our algorithm to fuse statements to enable array contraction appears in Figure 3. It takes as input an ASDG, G, and it returns a fusion partition $P = (P_1, P_2, \ldots, P_l)$ containing l clusters. Initially, P is the trivial fusion partition (line 1). The algorithm considers each variable,[2] x_i, that appears in the input array statement dependence graph in order of decreasing weight, $w(x_i, G)$. As a result, arrays that have potentially the largest single impact on the total contraction benefit are considered first. In line 5, set c is assigned all the fusible clusters that contain references to variable x_i. The fusion of all the statements in the fusible clusters in c might introduce inter-fusible-cluster cycles, so c becomes the union of itself and the fusible clusters that are on inter-fusible-cluster cycles using the GROW function (line 6). This guarantees that there will be no dependence cycles, for they prevent fusion. If variable x_i is contractible and a fusion partition is produced by combining all the fusible clusters in c (by Definitions 6 and 5), fusion is performed. The union of all fusible clusters in c is taken and assigned into the P_k with the smallest value k in c. The counter l is decremented to indicate that there are fewer clusters.

The FUSION-FOR-CONTRACTION algorithm uses three auxiliary routines. Function GROW(c, G) returns all fusible clusters not in c that are reachable by a dependence path from a cluster in c and that have a dependence path to a cluster in c. These are the fusible clusters that will be on

[2]For simplicity, we describe the algorithm as operating on array variables. In reality, it operates on array variable definitions, so that different references to the same array in disjoint live ranges can be optimized separately.

INPUT $G = (V, E)$: an array statement dependence graph

OUTPUT $P = (P_1, P_2, \ldots, P_l)$: a fusion partition of G

```
FUSION-FOR-CONTRACTION(G)
1   P ← trivial partition of G
2   l ← |V|
3   x ← array vars in G sorted by decreasing weight w
4   for i ← 1 to |x|   { consider var x_i for contraction }
5       c ← {P_j | P_j contains a reference to variable x_i}
6       c ← c ∪ GROW(c, G)
7       if CONTRACTIBLE?(x_i, c, G) and
              FUSION-PARTITION?(c, G)
8           k ← smallest j for P_j ∈ c
9           P_k ← ∪_{z∈c} z
10          l ← l − (|c| − 1)
11  return P
```

Figure 3: Algorithm to find a fusion partition that enables contraction in an ASDG.

an inter-fusible-cluster dependence cycle if the clusters in c are fused. This function's running time is $O(e)$, where e is the number of edges in G. The FUSION-PARTITION?(c, G) and CONTRACTIBLE?(x, c, G) predicates test the conditions in Definitions 5 and 6, respectively. They both run in $O(e)$ time. The former function can ignore inter-cluster cycles because line 6 guarantees they will not exist. It also calls FIND-LOOP-STRUCTURE (described in the next section) to decide whether condition (iv) of Definition 5 is met. If there are r arrays in G, the total running time for FUSION-FOR-CONTRACTION is $O(re)$.

The algorithm to perform fusion for locality enhancement is identical to that in Figure 3, except that the CONTRACTIBLE? predicate in line 7 is eliminated. We try to fuse all statements that reference the array that will have the greatest single locality benefit, which is analogous to the contraction benefit. Next, we will describe the process by which an ASDG is scalarized given a fusion partition.

4.2 Scalarization

Scalarization generates a loop nest for each fusible cluster in a fusion partition, where the loop nests and the statements in the loop nests are ordered by a topological sort using inter- and intra-fusible-cluster dependences, respectively. The only work remaining is deciding the structure of each loop nest, *i.e.*, the direction in which and dimension over which each loop iterates. This information is encoded in a loop structure vector (Definition 4) for each fusible cluster. Intra-cluster dependences constrain the structure of the loop nest that will implement its statements (*i.e.*, the loop nest must preserve these dependences). When the dependences do not fully constrain the structure of the loop nest, we will favor the loop structure that best exploits spatial locality.

The algorithm to find a loop structure vector given a set of unconstrained distance vectors from intra-fusible-cluster array-level dependences appears in Figure 4. FIND-LOOP-STRUCTURE consists of a doubly nested loop. The outer loop (line 3) iterates over the loops of the target loop nest, and the inner loop iterates over the dimensions of the arrays. The loop body matches loops to array dimensions (lines 7 through 11). We consider target loops from outer to inner because when a dimension is assigned to a loop, the dependences that are carried in that loop do not constrain the structure of the inner loops (thus set C is pruned in line 10).

```
INPUT C : a set of m unconstrained distance vectors, each of size n

OUTPUT p : a loop structure vector of size n (loop i iterates over
           array dimension |p_i| in the direction of the sign of p_i)

FIND-LOOP-STRUCTURE(C)
1  for j ← 1 to n               { initialize unassigned mask }
2      b_j ← true               { b_j = true ⇒ array dimension j has
                                 not yet been assigned to a loop}
3  for i ← 1 to n               { iterate over loops }
4      for j ← 1 to n               { iterate over array dimensions }
5          if b_j
                    ⎧ +1   if ∀u ∈ C, u_j ≥ 0
6              d ← ⎨ −1   if ∀u ∈ C, u_j ≤ 0 and ∃u ∈ C, u_j < 0
                    ⎩  0   otherwise
7              if d ≠ 0     { can loop i iterate over dimension j? }
8                  b_j ← false
9                  p_i ← jd
10                 C ← C − {u ∈ C | u_j ≠ 0}
11                 break out of j loop
12     return NOSOLUTION     { no dimension found for loop i }
13 return p
```

Figure 4: Algorithm to find a legal loop structure vector given a set of unconstrained distance vectors from intra-fusible-cluster data dependences.

```
B(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)                              (1)
C(1:n,1:m) = A(1:n,1:m)*A(1:n,1:m)

B(1:n,1:m) = A(0:n-1,1:m)+A(0:n-1,1:m)                          (2)
C(1:n,1:m) = A(1:n,1:m)*A(1:n,1:m)

B(1:n,1:m) = A(0:n-1,1:m)+C(0:n-1,1:m)                          (3)
C(1:n,1:m) = A(1:n,1:m)*A(1:n,1:m)

A(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)                              (4)

A(1:n,1:m) = A(0:n-1,1:m)+A(0:n-1,1:m)                          (5)

B(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)                              (6)
C(1:n,1:m) = B(1:n,1:m)

B(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)+C(0:n-1,1:m)                 (7)
C(1:n,1:m) = B(1:n,1:m)

T1(1:n,1:m) = B(1:n,1:m)
T2(1:n,1:m) = B(1:n,1:m)                                        (8)
A(1:n,1:m) = A(2:n+1,1:m) + T1(2:n+1,1:m) + T2(2:n+1,1:m)
```

Figure 5: Code fragments to exercise Fortran 90 and HPF compilers.

We consider dimensions from 1 to n so that inner loops will be matched with higher array dimensions to exploit spatial locality (assuming row-major allocation), if allowed by the constraints. If there are e dependences, the running time of lines 6 and 10 is $O(e)$, so FIND-LOOP-STRUCTURE runs in $O(n^2 e)$ time. Because the rank of the arrays, n, is typically very small and effectively constant [23], the algorithm is essentially linear, $O(e)$, in the number of dependences.

5 Evaluation

This section evaluates our algorithm for statement fusion and array contraction—as implemented in the ZPL compiler—by comparison to commercial F90/HPF compilers and hand coded C code. Furthermore, we examine the transformations' effect on memory use and their relative impact on runtime performance. Finally, we evaluate how their interaction with communication optimizations effect performance.

The benchmark programs we use to evaluate our transformations represent typical parallel array language programs. The SP application and EP kernel belong to the NAS parallel benchmark suite [2, 3]. SP solves sets of uncoupled scalar pentadiagonal systems of equations; it is representative of portions of CFD codes. EP generates pairs of Gaussian random deviates, and it is considered "embarrassingly parallel." EP characterizes the peak realizable FLOPS of a parallel machine. Tomcatv is a SPEC CFP95 benchmark that performs vectorized mesh generation. The Simple code solves hydrodynamics and heat conduction equations by finite difference methods [10]. The Fibro application uses mathematical models of biological patterns to simulate the dynamic structure of fibroblasts [11].

We use the Cray T3E, IBM SP-2 and Intel Paragon in our evaluation. The T3E is a distributed shared memory machine, while the other two are message passing distributed memory machines. The T3E we use has 94 nodes, each containing a 450 MHz DEC Alpha 21164, 8 and 96 KB L1 and L2 data caches, respectively, and 256 MB memory. The SP-2 we use has 144 nodes, each containing a 120 MHz POWER2 Super Chip (P2SC), 128 KB data cache and 256 MB memory. The Paragon we use has 18 nodes, each containing a 75MHz Intel i860 processor, 8 KB data cache and 32 MB of memory.

5.1 Comparison to Commercial Compilers

In order to assess the state of the art, we determine how aggressively current commercial array language compilers perform statement fusion and array contraction. We examine compilers for F90 and HPF (a parallel superset of F90) because F90 is the array language to which the greatest development effort has been devoted.

The developers of commercial compilers do not advertise the specific optimizations that their products perform, so we infer their ability to perform statement fusion and array contraction by studying compiler output for a set of carefully selected code fragments, shown in Figure 5. In all cases, arrays B, T1 and T2 are not live beyond the given code fragments. The fragments in (1), (2) and (3) test a compiler's ability to perform statement fusion to exploit temporal locality. The fragments differ in the data dependences they contain. The fragments in (4) and (5) test a compiler's ability to eliminate compiler temporaries, and (6) and (7) test the same for user temporaries, in this case array B. Fragment (8) contains two user arrays that can be contracted if contraction of the compiler array for the third statement is sacrificed. The fragment tests whether a compiler properly weighs this tradeoff. Figure 6 summarizes whether each compiler properly fused (and in some cases contracted) each code fragment.

First, observe that the PGI and IBM compilers appear not to perform any statement fusion (*i.e.*, each array statement compiles to a single loop nest). The implementors hoped to leverage the optimizations performed by the back end Fortran 77 compiler, which does in fact perform fusion. Unfortunately, the back end compiler does not perform contraction because it was not designed to compile scalarized array language programs. Most of the compilers successfully eliminate compiler temporaries. This is not surprising given that it requires only a simple local analysis, but additional experiments (Section 5.4) show that this trans-

compiler	*fusion* (1)	(2)	(3)	*compiler temps* (4)	(5)	*user temps* (6)	(7)	*trade-off* (8)
PGI HPF 2.1				√	√			
IBM XLHPF 1.2				√	√			
APR XHPF 2.0	√	√		√				
Cray F90 2.0.1.0	√	√		√	√	√		
ZPL 1.13	√	√	√	√	√	√	√	√

Figure 6: Observed behavior of five array language compilers. A √ indicates that the compiler produced the proper fused/contracted code (as described in the running text).

application	*array language* *w/o contr.*	*w/ contr.*	*% change*	*scalar lang.*
EP	22(0/22)	0(0/0)	-100.0	1
Frac	8(0/8)	1(0/1)	-87.5	1
SP	181(18/163)	56(0/56)	-69.1	48
Tomcatv	19(4/15)	7(0/7)	-63.2	7
Simple	85(20/65)	32(0/32)	-62.4	32
Fibro	49(0/49)	27(0/27)	-44.9	n/a

Figure 7: Static arrays contracted (categorized as compiler/user arrays). Fibro was developed in ZPL, so no equivalent scalar version exists.

formation alone is not sufficient. Though the APR compiler appears to performs fusion for locality and compiler array contraction, it is unable to fuse loops that carry anti-dependences.

Finally, notice that the Cray F90 compiler appears to perform both statement fusion and array contraction, but there are circumstances under which it fails. The compiler is unable to fuse statements where the resulting loop nest would contain loop carried anti-dependences. As a result, fusion does not occur in either (3) or (7), in the latter case inhibiting contraction. We also infer that the compiler considers contraction of compiler and user temporary arrays separately, since it contracts the compiler temporary in (8) at the expense of contracting the two user temporaries. The Cray compiler probably never inserts compiler temporaries when a single statement does not require it, even if this transformation would enable the contraction of multiple other arrays. The technique we describe always inserts compiler arrays, and it treats compiler and user arrays together as candidates for contraction. If a single statement does not truly require a compiler array, our algorithm is guaranteed to contract it unless a more favorable contraction is performed that prevents it.

5.2 Comparison to Hand-coded

A successful array language compiler will produce scalar code comparable to that of a skilled scalar language programmer. We now compare code produced by the ZPL compiler with equivalent programs written in a scalar language. Figure 7 summarizes for each of the six benchmarks the number of static arrays appearing in the compiled code with and without array contraction. Note that within each code, nearly all arrays are approximately the same size. We see that all compiler-generated arrays have been eliminated. The benefit of this is that a programmer can better comprehend the memory use of their code when the compiler only infrequently introduces arrays. Figure 7 shows a substantial reduction in the number of static arrays. All the arrays are eliminated in EP, and in all but one of the other benchmarks more than half are eliminated.

The final column in Figure 7 gives the number of arrays that appear in equivalent scalar language codes. The scalar language codes are all publicly available C or Fortran 77 programs written by third parties. The compiler-generated code has the same or fewer arrays on all the benchmarks except SP, which highlights a deficiency in our current algorithm. As we have described contraction, an array is contracted to a scalar or left as is. SP contains a great many opportunities to contract arrays to lower dimensional arrays. Though the resulting arrays cannot be manipulated in registers, they conserve memory and make better use of the cache. Despite this shortcoming, SP still benefits from a substantial performance improvement, as we see in Section 5.4.

5.3 Effect on Memory Usage and Problem Size

While the preceding section uses static array counts to suggest that contraction conserves memory, here we employ dynamic data to discover more precisely how memory conservation from array contraction enables larger problems to be solved in a fixed amount of memory. The degree by which contraction allows larger problems to be solved is an important issue for memory bound applications. We assume the following of a single program on a particular machine: (i) all arrays are the same size, which we call the *problem size*, (ii) all array elements are the same size, and (iii) a constant amount of memory is available for array allocation independent of problem size. The degree by which the maximum problem size scales due to contraction is the ratio of the maximum problem size after and before contraction, $\frac{s_a}{s_b}$. Given the above assumptions and that maximum problem size is inversely proportional to the maximum number of simultaneously live arrays, l, the scaling factors becomes $\frac{l_b}{l_a}$. We subtract 1 and multiply by 100 to convert the maximum problem size scaling factor to percent change,

$$C(l_b, l_a) = 100 \times \frac{l_b - l_a}{l_a}.$$

The first columns of Figure 8 give the dynamic l_b and l_a values and the calculated C value for each benchmark.

To confirm the above analysis, we experimentally determine for each benchmark the largest problem size that fits on a single node of the Cray T3E and the IBM SP-2. Both machines have operating system facilities to limit the process size, so we found the largest problem size that does not result in a memory allocation failure. Columns seven and ten of Figure 8 give the change in problem size, both along one dimension of the problem domain and in total data volume. The experimental data shows that these applications respect the above assumptions, for the C value accurately predicts the change in problem volume. The one exception is Frac on the SP-2, which violates assumption (ii). EP, in which all arrays are eliminated, clearly benefits the most from contraction because the contracted form uses a constant amount of memory, independent of the problem size. The other applications' changes in problem size vary from 10% to 274% along a single dimension or 25% to 1300% in total volume.

application	l_b	l_a	C	IBM SP-2 maximum problem size w/o contr.	w/ contr.	% change (vol)	Cray T3E maximum problem size w/o contr.	w/ contr.	% change (vol)
EP	22	0	∞	2^{19}	∞	$\infty(\infty)$	2^{16}	∞	$\infty(\infty)$
Frac	8	1	700.0	1531^2	5730^2	274.3(1300.7)	1409^2	3987^2	183.0(700.7)
Tomcatv	19	7	171.4	929^2	1530^2	64.7(171.2)	1293^2	2128^2	64.6(170.9)
Fibro	49	27	81.5	583^2	790^2	35.5(83.6)	572^2	774^2	35.3(83.1)
SP	23	17	35.3	74^3	81^3	9.5(31.1)	91^3	101^3	11.0(37.7)
Simple	40	32	25.0	640^2	715^2	11.7(24.8)	623^2	702^2	12.7(27.0)

Figure 8: Effect of contraction on maximum achievable problem size on single IBM SP-2 and Cray T3E nodes.

5.4 Run-time Performance

This section considers the runtime performance impact of array contraction and statement fusion. Though we discuss only the relative effect of these transformations, other studies have shown that the ZPL compiler produces code that performs within 10% of hand coded C plus message passing and generally better than HPF [8, 17, 18, 20].

In order to better understand the performance contributions of fusion and contraction, we measure execution time using several incrementally different optimization strategies.

baseline : no fusion or contraction transformations are performed

f1 : fusion is performed to enable the contraction of compiler arrays, but contraction is not performed

c1 : fusion is performed to enable the contraction of compiler arrays, and contraction is performed

f2 : *c1* plus fusion is performed to enable contraction of user arrays, but the contraction is not performed

f3 : *c1* plus fusion is performed to improve locality (as described in Section 4)

c2 : *c1* plus fusion is performed to enable contraction of user arrays, and contraction is performed

c2+f3 : *c2* plus fusion is performed to improve locality (as described in Section 4)

c2+f4 : *c2+f3* plus all legal fusion (by a greedy pair-wise algorithm)

Figures 9, 10 and 11 show the percent improvement of each transformation over *baseline* for each benchmark for a varying number of processors on the Cray T3E, IBM SP-2 and Intel Paragon. Execution times are the best of three trials on the T3E and Paragon and of at least six trials on the SP-2, a machine that suffers from great performance variance from trial to trial. So that we may neutralize the effect of communication masking all other performance characteristics on large processor sets, we scale the problem sizes with the number of processors (*i.e.*, the amount of data per processor remains constant as the number of processors increases).

These graphs demonstrate that performing contraction on both compiler and user arrays in array languages is essential. The predominant characteristic of the graphs is that *c2* dominates the other transformations. The elimination of a large portion of the compiler and user arrays by contraction drastically improves temporal locality, always resulting in a significant performance boost (up to 400% on one application). Fibro on the SP-2 does not benefit from contraction for large number of processors because of interactions with communications optimizations discussed in the next section. In the larger applications, contraction of only compiler arrays, *c1*, provides a substantive performance enhancement (up to 30%), but it is only a fraction of the potential contraction benefit. The smaller benchmarks, such as Fibro, EP and Frac, require no compiler arrays, so they do not benefit from *f1* and *c1*. Clearly, transformation *c1* does not sufficiently address the problem of unnecessary temporary arrays in array languages.

For a number of programs, transformations *f2* and *f3* produce noticeable slowdown. It appears that they increase capacity and conflict misses in programs that are particularly sensitive to memory system performance, such as Tomcatv and Fibro. Transformation *c2+f4* generally results in no improvement beyond *c2+f3*, and frequently produces significantly less improvement versus baseline (3% versus 16% for Fibro on the T3E). SP is the one exception, because arbitrary fusion enhances spatial locality of independent statements. Our fusion algorithm instead fuses dependent statements to enhance temporal locality. We leave to future work the extension of our algorithm for spatial locality sensitivity. The lesson is that fusion should not be performed arbitrarily in an array language.

As the number of processors, p, varies, certain trends become evident. The improvement due to contraction in EP and Frac is effectively independent of the number of processors because these codes scale nearly perfectly with p. The improvement due to fusion and contraction grows with p for some programs, such as Simple and Tomcatv on the SP-2, when the transformations improve portions of the program that make up a larger fraction of total execution time as p grows (*i.e.*, the transformations improve portions of the code that do not scale well with p).

The performance improvement for a transformation decreases with p when the transformation improves a portion of the code that makes up a smaller fraction of total execution time as p increase. This happens when some other segment of the code is not scaling well and consumes a larger fraction of total execution time as p increases. SP exhibits this behavior because only potions of the code that scale well benefit from the transformations. When both scaling and non-scaling segments of a code benefit from the transformations, machines characteristics (*e.g.*, the relative costs of cache misses, communication and floating point operations) dictate the trends. This is exemplified by Tomcatv, which shows level, increasing and decreasing trends on the three machines in our experiments.

5.5 Interaction with Communication Optimization

In this section, we demonstrate that statement fusion interacts with communication optimizations and for this reason should be performed at the array level. Some optimizations cannot be performed practically at the scalar level because

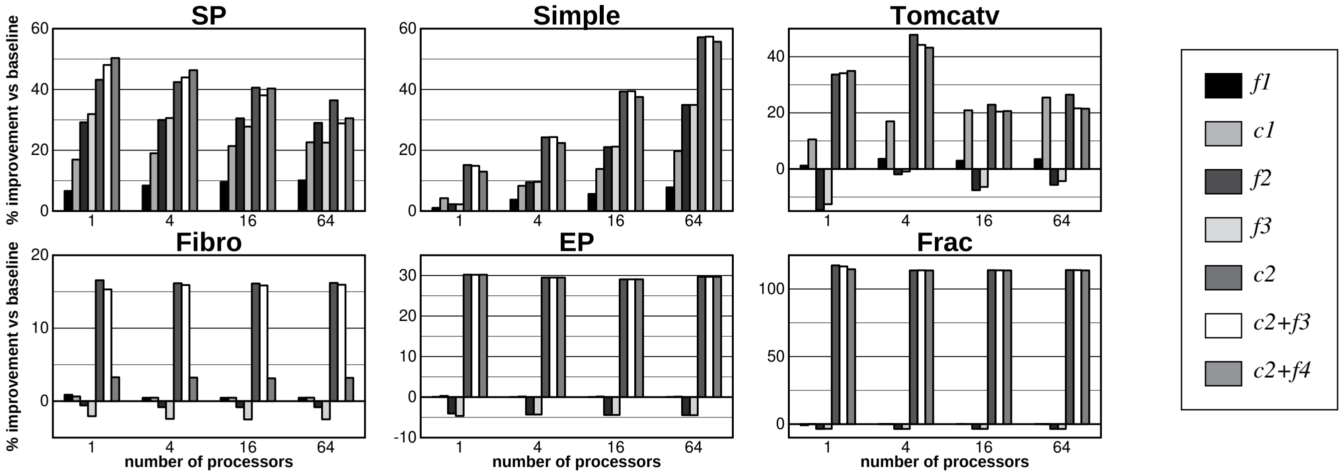


Figure 9: Benchmark performance on Cray T3E. Negative bars represent slowdown.

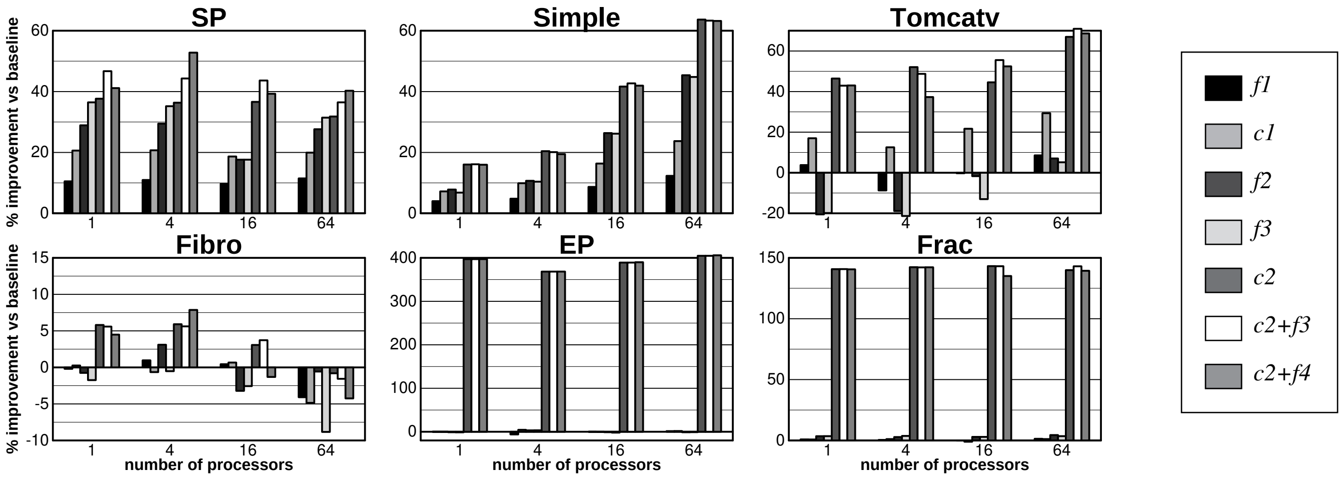


Figure 10: Benchmark performance on IBM SP-2. Negative bars represent slowdown.

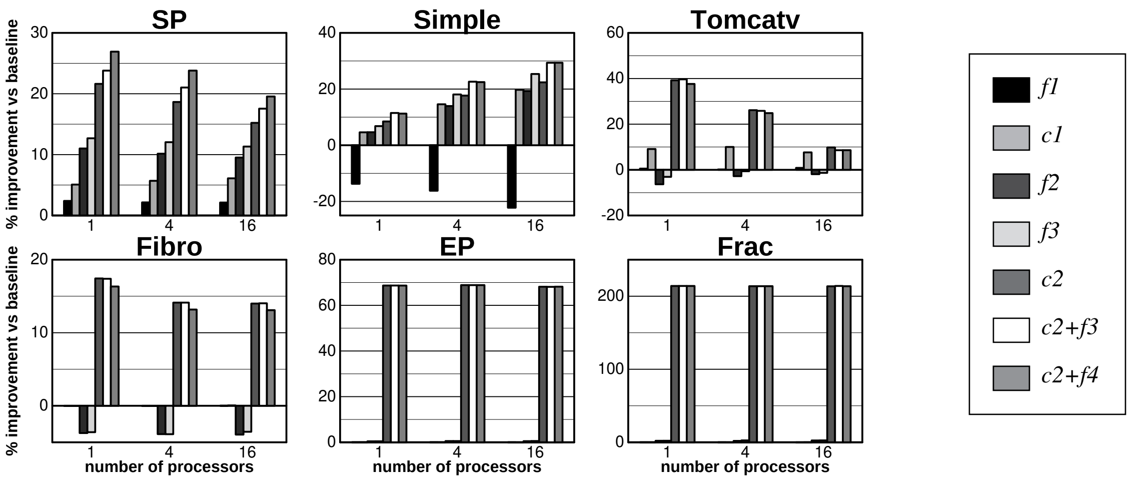


Figure 11: Benchmark performance on Intel Paragon. Negative bars represent slowdown.

they interact with other transformations that can only occur at the array level. If an optimization that interacts with array level transformations is relegated to a scalar compiler, either the array level transformations must understand and reason about the optimization behavior of the scalar compiler or vice versa. It is unlikely that scalar compilers can understand the optimization strategy of all the compilers that compile to it, so the array compiler must consider scalar optimizations when performing array transformations, effectively moving the scalar transformations into the array compiler.

To achieve efficient parallel execution, compilers must often perform aggressive communication optimizations [9], such as redundancy elimination, message combining and pipelining. In some cases, these communication optimizations are at odds with fusion for contraction. For example, pipelining hides latency by separating the send and receive portions of communication with computation, but fusion may collect into a single loop some of the statements that could be used to hide latency, potentially disabling overlap. The experiments presented thus far resolve this conflict by favoring fusion, *i.e.*, fusion is never prevented by communication optimizations. We consider an alternative strategy in which communication optimizations are favored, *i.e.*, fusion cannot be performed if it reduces the benefit of communication optimization. Message vectorization never conflicts with fusion, so it is always performed.

As the amount of fusion increases, the potential for conflict with communication optimization grows. The preceding section demonstrates that *c2+f4* is not a valuable transformation, so we use the *c2+f3* transformation. On the T3E, when favoring communication optimizations over fusion for contraction, Simple, Tomcatv, SP and Fibro suffer a slowdown of 25.4%, 22.7%, 9.6% and 5.1%, respectively. On the SP-2, they slowdown by 31.8%, 66.5%, 10.5% and -10.6%, respectively. On the Paragon, they slowdown by 7.5%, 8.5%, 5.0% and 0.9%. The first three programs slowdown significantly because the communication optimizations disable a large number of array contraction opportunities without producing comparable communication benefits. Only one fusion for locality opportunity and no contraction opportunities are lost by favoring communication optimizations in Fibro. It slows down little and in one case it speeds up, because of the additional communication optimization. EP and Frac do not slowdown because they are small codes that do not benefit from communication optimization, with or without fusion.

We have not demonstrated that favoring contraction is optimal, but we have shown that if a choice is to be made, fusion for contraction should be favored. This suggests that it would be very difficult to perform communication optimizations if fusion and contraction occur after scalarization. The communication transformations would have to understand contraction well enough to optimize without disabling it, since it is unlikely that the scalar compiler could reason about communication primitives once they are scalarized. The Fibro data suggests that there are delicate tradeoffs that only an integrated approach to fusion and communication optimization can address, which would further complicate performing fusion at the scalar level. Furthermore, we expect to find that integration will become even more important on machines with low cost synchronization in hardware (*e.g.*, SGI Origin, Sun E10000). Thus, these results support our claim that these optimizations for array languages should be performed at the array level.

6 Related Work

The problem of optimizing array languages at the array level has recently received attention by others. Hwang *et al.* describe a scheme for *array operation synthesis* [14]. Multiple instances of element-wise F90 array operations such as MERGE, CSHIFT, and TRANSPOSE are combined into a single operation, reducing data movement and intermediate storage. Their work does not address the inter-statement intermediate array problem except to substitute an intermediate array's use by its definition. This *statement merge* optimization [15] enables more operation synthesis, but it is not always possible, and it potentially introduces redundant computation and increases overall program execution time. Roth and Kennedy have independently developed a similar array based data dependence representation for F90, and they describe its use in scalarization [21]. They do not address the fusion for contraction problem.

Loop fusion in the context of scalar programming languages such as Fortran 77 is well understood [26]. Though most work only considers pairwise fusion, some research addresses collective loop fusion, as we do. Sarkar and Gao [22] transform loop nests by loop reversal, interchange and fusion to enable array contraction. They target multiprocessors and exploit pipelining by executing producer and consumer loops on different processors, so they are free to ignore all but flow dependences. Because we instead distribute iteration spaces, preservation of all types of dependences is critical to our solution. Gao *et al.* [12] describe another technique for loop fusion based on a maxflow algorithm. The technique requires its input loop nests to be identically controlled, and it does not perform loop reversal nor interchange to enable additional fusion. Furthermore, it is unclear what the algorithm does when a potentially contractible array is consumed by multiple loop nests. Our collective scheme performs reversal, interchange and fusion simultaneously to enable contraction.

Carr and Kennedy recognized the importance of keeping array values in scalars through *scalar replacement* [4], which is similar to array contraction in that some array references become scalar references, but array allocation is not eliminated (*i.e.*, memory usage is not reduced). Their focus is in recognizing the opportunity in a scalar loop nest, while ours is in enabling the opportunity in an array language compiler via statement fusion.

Many techniques for improving locality by loop transformations have appeared in the literature [5, 16, 19, 25]. Much of this work addresses the issue of managing the conflicting goals of improving locality without sacrificing parallelism. This is a far less important issue in an array language compiler, for the compiler can assume that only the loops that it generates need to be parallelized; user loops can remain sequential. In this paper we have assumed that all dimensions of all arrays are distributed and are a potential source of parallelism.

7 Conclusion

This paper has shown how statement fusion can be performed at the array level to enable array contraction and to enhance locality. We have introduced, for array statements, a normal form and data dependence machinery that leverages array language properties. We have empirically demonstrated that our array-level transformations produce substantial performance improvements, both in execution time and in memory usage. We have found that the common technique of only contracting compiler arrays is insufficient for achieving high performance. Finally, we have shown that fusion and contraction should be performed at the array level, *i.e.*, before scalarization, because they must precede or be integrated with communication optimizations for best performance.

Acknowledgments. We thank Sung-Eun Choi, Bradford Chamberlain, Jerry Roth, and the anonymous reviewers for their comments and insights on early drafts of this paper. This research was supported by a grant of HPC time from the Arctic Region Supercomputing Center. This research was conducted using the resources of the Cornell Theory Center, which receives major funding from the National Science Foundation and New York State, with additional support from the National Center for Research Resources at the National Institutes of Health, IBM Corporation, and other members of the center's Corporate Partnership Program.

References

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

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks (94). Technical report, RNR Technical Report RNR-94-007, March 1994.

[3] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical report, NAS Report NAS-95-020, December 1995.

[4] Steve Carr and Ken Kennedy. Scalar replacement in the presence of conditional control flow. *Software – Practice and Experience*, 24(1):51–77, January 1994.

[5] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improved data locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994. San Jose, CA.

[6] B. Chamberlain, S. Choi, E Lewis, C. Lin, L. Snyder, and W. D. Weathersby. Factor-join: A unique approach to compiling array languages for parallel machines. In David Sehr, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Proceedings of the Ninth International Workshop on Languages and Compilers for Parallel Computing*, pages 481–500. Springer-Verlag, August 1996.

[7] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL's WYSIWYG performance model. To appear in *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.

[8] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. The case for high level parallel programming in ZPL. To appear in *IEEE Computational Science and Engineering*, 1998.

[9] Sung-Eun Choi and Lawrence Snyder. Quantifying the effect of communication optimizations. In *International Conference on Parallel Processing*, August 1997.

[10] W. Crowley, C. P. Hendrickson, and T. I. Luby. The SIMPLE code. Technical Report UCID-17715, Lawrence Livermore Laboratory, 1978.

[11] Marios D. Dikaiakos, Calvin Lin, Daphne Manoussaki, and Diana E. Woodward. The portable parallel implementation of two novel mathematical biology algorithms in ZPL. In *Ninth International Conference on Supercomputing*, 1995.

[12] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295. Springer-Verlag, 1992.

[13] High Performance Fortran Forum. *High Performance Fortran Langauge Specification, Version 2.0.* January 1997.

[14] Gwan Hwan Hwang, Jenq Keun Lee, and Dz Ching Ju. An array operation synthesis scheme to optimize Fortran 90 programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[15] Dz-Ching Ju. *The Optimization and Parallelization of Array Language Programs.* PhD thesis, University of Texas–Austin, August 1992.

[16] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *International Conference on Supercomputing*, pages 323–334, July 1992.

[17] C. Lin, L. Snyder, R. Anderson, B. Chamberlain, S. Choi, G. Forman, E. Lewis, and W. D. Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report 95–11–05, Department of Computer Science and Engineering, University of Washington, 1994.

[18] Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Workshop on Languages and Compilers for Parallel Computing*, pages 361–375. Springer-Verlag, 1994.

[19] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE transactions on parallel and distributed systems*, 8(2):193–209, February 1997.

[20] Ton A. Ngo. *The Role of Performance Models in Parallel Programming and Languages.* PhD thesis, University of Washington, Department of Computer Science and Engineering, 1997.

[21] Gerald Roth and Ken Kennedy. Dependence analysis of Fortran90 array syntax. In *Proceedings of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, August 1996.

[22] Vivek Sarkar and Guang R. Gao. Optimization of array accesses by collective loop transformations. In *International Conference on Supercomputing*, pages 194–205, June 1991.

[23] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[24] Lawrence Snyder. *Programming Guide to ZPL.* MIT Press, 1998. to appear.

[25] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN'91 Conference on Program Language Design and Implementation*, June 1991.

[26] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.