An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs

Gwan-Hwan Hwang Jenq Kuen Lee
Department of Computer Science, National Tsing-Hua University, Taiwan

Dz-Ching Ju
Hewlett-Packard Company, Cupertino, CA 95014, USA

Abstract

An increasing number of programming languages, such as Fortran 90 and APL, are providing a rich set of intrinsic array functions and array expressions. These constructs which constitute an important part of data parallel languages provide excellent opportunities for compiler optimizations. In this paper, we present a new approach to combine consecutive data access patterns of array constructs into a composite access function to the source arrays. Our scheme is based on the composition of access functions, which is similar to a composition of mathematic functions. Our new scheme can handle not only data movements of arrays of different numbers of dimensions and segmented array operations but also masked array expressions and multiple sources array operations. As a result, our proposed scheme is the first synthesis scheme which can synthesize Fortran 90 RESHAPE, EOSHIFT, MERGE, and WHERE constructs together. Experimental results show speedups from 1.21 to 2.95 for code fragments from real applications on a Sequent multiprocessor machine by incorporating the proposed optimizations.

1 Introduction

An increasing number of programming languages, such as Fortran 90[1], HPF (High Performance Fortran)[11], APL, and possibly C++ with built-in array classes[7], provide a rich set of intrinsic array functions and array parallelism expressions. Those intrinsic array functions and array parallelism expressions can directly operate on the elements of a multi-dimensional array object simultaneously without requiring iterative statements. They can efficiently support parallel execution and improve the portability of programs by explicitly exposing the data parallelism of each operation.

Multiple consecutive array operations specify a particular mapping relationship between the source arrays and final target array. A straightforward compilation for these consecutive array functions or array expressions may translate each array operation into a (parallel) nested loop and use a temporary array to pass intermediate results to following array functions. The synthesis of multiple consecutive array functions or array expressions can compose several data access functions into an equivalent composite reference pattern. Thus, the synthesis can improve performance by reducing redundant data movement, temporary storage usage, and parallel loop synchronization overhead.

For example, consider the array expression below which consists of several array functions.

```
B=CSHIFT(( TRANSPOSE(EOSHIFT(A,1,/"0"/,2)) +
           RESHAPE(C,/4,4/) ),1,1)
```

The goal of array operation synthesis is to find a function $\mathcal{F}'$ at compiler time such that $B = \mathcal{F}'(\mathcal{A}, \mathcal{C})$ can be used to substitute the original expression above. Similarly, we can use expression, $A = \mathcal{F}'(\mathcal{X})$ to replace the expression below

```
A=CSHIFT(CSHIFT(TRANSPOSE(CSHIFT(CSHIFT(
          TRANSPOSE(X),1,+1),2,+1)),1,-1),2,-1)
```

where $\mathcal{F}'$ is the identity function.

In this paper, we propose a new array operation synthesis scheme to find the synthesized function, $\mathcal{F}'$ above. Our scheme is based on the composition of mathematic functions. It starts with preparing a mathematic access function for each kind of array expression or intrinsic array function. We then use a function composition approach to compose those mathematic access functions. This composition is semantically equivalent to the synthesis of array functions. Our scheme can handle data accesses between arrays of different numbers of dimensions, masked array expressions, multiple sources array

[0]Hwang and Lee's work was supported in part by NSC of Taiwan under grant no. NSC842213E007014 and NSC842221E007005.

operations, and segmented array functions. Although the proposed scheme is applicable to any programming language with array expression syntax and intrinsic array functions, we are particularly interested in optimizing Fortran 90 array language programs. As a result, we can handle compositions of extensive Fortran 90 array constructs, such as RESHAPE, SPREAD, EOSHIFT, TRANSPOSE, CSHIFT, and MERGE functions, as well as the synthesis of WHERE and ELSE-WHERE constructs. None of the existing synthesis scheme can synthesize Fortran 90 RESHAPE, SPREAD, MERGE, and WHERE constructs together.

Previously, the synthesis of array functions has been done mainly by using a stepper mechanism. Guibas and Wyatt[4] first proposed a universal selector representation called a stepper to encode the way in which elements of the current function partake in the formation of the final target. The data structure in a stepper consists of four arrays which specify the corresponding axis, start, and stride in the source array and also the size of each target axis. Budd[3] presented several variant data request representations: arithmetic progression vector, column vector, and offset vector, which aimed at vector machines. Ju, Wu, and Carini[5] proposed a scheme which is derived from the stepper representation, but they considered a Catenate function which was not treated in [4]. The data access pattern of each function is characterized by a stepper representation and a subarray boundary. The stepper mechanism, though heavily used by previous researchers, can not handle masked array expressions and array operations of multiple sources effectively. As a consequence, the stepper mechanism fails to synthesize the Fortran 90 constructs, such as MERGE function and WHERE construct. In addition, the variants of steppers are weak either in specifying the data accesses between arrays of different dimensional space[5] or in supporting segmented array functions[3].

The remainder of the paper is organized as follows. Section 2 presents the classification of array operations which is the foundation to our array operation synthesis. Section 3 then presents the composition method to synthesize array operations. Section 4 discusses the optimization of index ranges generated by the synthesis scheme. segmentation descriptors into loop bound checking. Section 5 extends the method to handle the Fortran 90 WHERE and ELSE-WHERE constructs. Section 6 discusses a system anomaly and proposes a solution. Section 7 shows experimental results. Finally, Section 8 concludes this paper.

2 Classification of Array Operations

In this section, we classify the array operations considered in this paper into four types based on their **data access functions**, which specify the mapping between the elements of the source and target arrays. The classification is listed in the following table:

	continuous	segmented
single-source	(1)	(3)
multiple-source	(2)	(4)

TYPE 1: single-source continuous data access function

A type 1 array operation has only one source array, and the array movement is continuous.

Definition 1 Consider an n-dimensional target array T and an m-dimensional source array S. A **type 1 array operation** has the following data access function:

$$\mathrm{T}[i_1,i_2,\cdots,i_n]=\mathrm{S}[f_1(i_1,i_2,\cdots,i_n),\ f_2(i_1,i_2,\cdots,i_n),\cdots,\ f_m(i_1,i_2,\cdots,i_n)]$$

where f_i is an index function.

The index functions from f_1 to f_n specify the array subscripts of source array S at the right-hand side (RHS). The array subscript expression at the left-hand side (LHS) of the data access function are index variables. Assume $\forall\ i,\ 1 \le i \le n$, r_i is the size of the i-th dimension of T. A type 1 array operation can be translated into the following pseudo code:

```
FORALL(i1=1;i1≤r1;i1++)
    FORALL(i2=1;i2≤r2;i2++)
        ...
            FORALL(in=1;in≤rn;in++)
                T[i1,i2,...,in]=S[f1(i1,i2,...,in), f2(i1,i2,...,in),...,
                                  fm(i1,i2,...,in)]
```

The TRANSPOSE function is an example of a type 1 array operation. Assume A and B are two-dimensional arrays. The data access function of B=TRANSPOSE(A) is B[i,j]=A[j,i] where f_1(i,j)=j and f_2(i,j)=i.

TYPE 2: multiple-source continuous data access function

Definition 2 Consider an n-dimensional target array T and k source arrays, $S_1,\cdots,S_k$. Let d_i be the number of the dimension of array S_i. A **type 2 array operation** has the following data access function:

$$\begin{aligned}&\mathrm{T}[i_1,i_2,\cdots,i_n]=\\ &\quad\mathcal{F}\ (\ S_1[f_1^1(i_1,i_2,\cdots,i_n),\ f_1^2(i_1,i_2,\cdots,i_n),\ \cdots,\ f_1^{d1}(i_1,i_2,\cdots,i_n)],\\ &\qquad\cdots,\\ &\qquad S_k[f_k^1(i_1,i_2,\cdots,i_n),\ f_k^2(i_1,i_2,\cdots,i_n),\ \cdots,\ f_k^{dk}(i_1,i_2,\cdots,i_n)]\\ &\qquad)\end{aligned}$$

where each f_i^j is an index function similar to that in the type 1 definition, and $\mathcal{F}$ is a k-ary function called calculation function.

Assume $\forall\ i,\ 1 \le i \le n$, r_i is the size of the i-th dimension of T. In general, a type 2 array operation can be translated into the following template.

FORALL(i_1=1:$i_1 \le r_1$: i_1++)
 FORALL(i_2=1:$i_2 \le r_2$:i_2++)
  ...
  FORALL(i_n=1:$i_n \le r_n$:i_n++)
   T[i_1,i_2,$\cdots$,i_n] =
    $\mathcal{F}(S_1[f_1^1(i_1,i_2,\cdots,i_n),\ f_1^2(i_1,i_2,\cdots,i_n),\ \cdots,\ f_1^{d_1}(i_1,i_2,\cdots,i_n)]$,
      $S_2[f_2^1(i_1,i_2,\cdots,i_n),\ f_2^2(i_1,i_2,\cdots,i_n),\ \cdots,\ f_2^{d_2}(i_1,i_2,\cdots,i_n)]$,
      ...
      $S_k[f_k^1(i_1,i_2,\cdots,i_n),\ f_k^2(i_1,i_2,\cdots,i_n),\ \cdots,\ f_k^{d_k}(i_1,i_2,\cdots,i_n)]$
)

The data access function for C(:,:)=A(:,:)+B(:,:) is C[i,j]=$\mathcal{F}$(A[i,j],B[i,j)] where $\mathcal{F}$(x,y)=x+y. A more complicated example is the MERGE function in Fortran 90. The data access function for R=MERGE(T,F,M) is R[i,j]=$\mathcal{F}$(T[i,j],F[i,j],M[i,j]) where

$$\mathcal{F}(x,y,z) = \begin{cases} x & \text{if } z = true \\ y & \text{if } z = false \end{cases}$$

TYPE 3: single-source segmented data access function

For certain array operations, the data accesses can not be represented in a single continuous pattern, but instead they can be represented in multiple access patterns, each having an array index range. We use a **segmentation descriptor** to represent an array index range. A segmentation descriptor, ϕ, is a boolean predicate of the form, $\phi(/f_1(i_1,i_2,\cdots,i_n),\cdots,f_m(i_1,i_2,\cdots,i_n)/,\ /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$, where f_i is an index function, and l_i, u_i, and s_i are the lower bound, upper bound, and stride of f_i, respectively. The default value of s_i is one. If α_1 and α_2 are two segmentation descriptors, $(\alpha_1 \vee \alpha_2)$ and $(\alpha_1 \wedge \alpha_2)$ are also segmentation descriptors. A segmentation descriptor is **simple** if it consists of only a boolean function without "$\vee$" and "$\wedge$" operators.

Definition 3 Consider an n-dimensional target array T and an m-dimensional source array S. A **type 3 array operation** has the following data access function:

T[i_1,i_2,$\cdots$,i_n] =

$$\begin{cases} S[f_1(i_1,i_2,\cdots,i_n),\ f_2(i_1,i_2,\cdots,i_n),\ \cdots,\ f_m(i_1,i_2,\cdots,i_n)] & | \quad \gamma_1 \\ S[g_1(i_1,i_2,\cdots,i_n),\ g_2(i_1,i_2,\cdots,i_n),\ \cdots,\ g_m(i_1,i_2,\cdots,i_n)] & | \quad \gamma_2 \\ \cdots \\ \cdots \end{cases}$$

where γ_i is a segmentation descriptor to specify the index range for the mapping before the vertical bar.

For example, assuming that A and B are 4 by 4 matrices, the data access function of B=CSHIFT(A,1,1) is

$$B[i,j] = \begin{cases} A[i-3,j] \mid \phi(/\text{i,j}/,/4{:}4,1{:}4/) \\ A[i+1,j] \mid \phi(/\text{i,j}/,/1{:}3,1{:}4/) \end{cases}$$

TYPE 4: multiple-source segmented data access function

Definition 4 Consider an n-dimensional target array T and k source arrays, $S_1,\cdots,S_k$. Let d_i be the number of dimension of array S_i. A **type 4 array operation** has the following data access function:

T[i_1,i_2,$\cdots$,i_n]=

$$\begin{cases} \mathcal{F}1[S_1(f_1^1(i_1,i_2,\cdots,i_n),\cdots,f_1^{d1}(i_1,i_2,\cdots,i_n)],\cdots, \\ \quad S_k[f_k^1(i_1,i_2,\cdots,i_n),\cdots,f_k^{dk}(i_1,i_2,\cdots,i_n)]) \mid \gamma_1 \\ \mathcal{F}2[S_1(h_1^1(i_1,i_2,\cdots,i_n),\cdots,h_1^{d1}(i_1,i_2,\cdots,i_n)],\cdots. \\ \quad S_k[h_k^1(i_1,i_2,\cdots,i_n),\cdots,h_k^{dk}(i_1,i_2,\cdots,i_n)]) \mid \gamma_2 \\ \cdots \\ \cdots \end{cases}$$

Although no array function in Fortran 90 belongs to type 4, the composition of multiple array functions may result in a type 4 data access function. Appendix B gives a complete summary of data access functions for Fortran 90 array operations which we can handle in this paper.

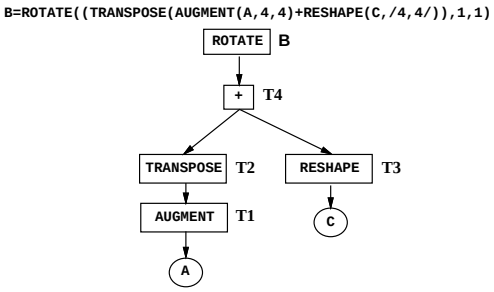

Figure 1: Parse tree of an array expression

3 Synthesis of Array Operations

Our approach to perform array operation synthesis is divided into four steps:

Step 1: Build the parse tree of the target array expression

First, we construct a parse tree for an array expression. Figure 1 shows the parse tree for the expression,

B=CSHIFT((TRANSPOSE(AUGMENT[1](A,4,4)) + RESHAPE(C,/4,4/)),1,1)

The source arrays are in the leaf nodes. In this example, arrays A and C are the source arrays. Each internal node including the root node corresponds to an array operation. We arbitrarily assign a unique name to each temporary array and attach the name to the node which will put its intermediate result in the temporary array in a straightforward translation. The array labeled as the root node is the target array and will contain the final result of the array expression upon completion.

Step 2: Prepare the data access functions

In this step, we derive the data access function for each internal node. We continue to use the example in Figure 1. For the AUGMENT function, the access function is

[1] AUGMENT function expands an array into a bigger size array by filling zero in the expanded region. AUGMENT function is not a standard Fortran 90 array function. We consider it because it inherently needs to have a non-trivial segmentation descriptor.

$$T1[i,j] = \begin{cases} A[i,j] & | \quad \phi(/\text{i,j}/,/1{:}3,1{:}3/) \\ 0 & | \quad \phi(/\text{i,j}/,/4{:}4,1{:}4/) \vee \phi(/\text{i,j}/,/1{:}4,4{:}4/). \end{cases}$$

The access function for the TRANSPOSE function is T2[i,j]=T1[j,i], for the RESHAPE function is T3[i,j]=C[i+j*4], and for the "+" operation is T4[i,j]=$\mathcal{F}$(T2[i,j],T3[i,j]), where $\mathcal{F}$(x,y)=x+y. The access function for the CSHIFT function is

$$B[i,j] = \begin{cases} T4[i+1,j] & | \quad \phi(/\text{i,j}/,/1{:}3,1{:}4/) \\ T4[i-3,j] & | \quad \phi(/\text{i,j}/,/4{:}4,1{:}4/). \end{cases}$$

Step 3: Synthesize data access functions

In this step, the collected data access functions in step 2 are synthesized to one data access function. We start with the data access function in the root node of the parse tree. For each temporary array at the RHS of the data access function, we substitute it with the data access function of that temporary array. The substitution may produce other temporary arrays in the RHS. This substitution processes is repeated until all the temporary arrays in the RHS are substituted by the source arrays.

A data access function may contain multiple access patterns delimited by segmentation descriptors. For different access patterns, we classify the substitution process into 4 cases: (1) two consecutive single-source data access patterns, (2) a multiple-source data access pattern followed by a single-source data access pattern, (3) a single-source data access pattern followed by a multiple-source data access pattern, (4) two consecutive multiple-source data access patterns. In the case that a access pattern has no segmentation descriptor, we treat the outcome of its segmentation descriptor as always true. In addition, substitution process has to be done with each access pattern of access functions. We describe these four cases below.

CASE 1:

The following mapping shows two consecutive single-source data access patterns:

$$T[i_1,i_2,\cdots,i_n] = S[\ f_1(i_1,i_2,\cdots,i_n),\ f_2(i_1,i_2,\cdots,i_n),\cdots, f_m(i_1,i_2,\cdots,i_n)\ ] \mid \gamma$$

$$S[i_1,i_2,\cdots,i_m] = Q[\ h_1(i_1,i_2,\cdots,i_m),\ h_2(i_1,i_2,\cdots,i_m),\cdots, h_p(i_1,i_2,\cdots,i_m)\ ] \mid \beta$$

where f_i *and* h_j are the index functions, γ and β are segmentation descriptors, and

$\beta = \phi(/i_1,i_2,\cdots,i_m/, /l_1 : u_1 : s_1, l_2 : u_2 : s_2, \cdots, l_m : u_m : s_m/)$.

The S in the RHS of the first access pattern can be replaced with Q with respect to the second access pattern. It yields the following access function:

$$T(i_1,i_2,\cdots,i_n) = Q(\ g_1(i_1,i_2,\cdots,i_n),\ g_2(i_1,i_2,\cdots,i_n), \cdots,\ g_p(i_1,i_2,\cdots,i_n)\ ) \mid \gamma \wedge \beta'$$

where $g_i(i_1,i_2,\cdots,i_n) = h_i(f_1(i_1,i_2,\cdots,i_n),\cdots, f_m(i_1,i_2,\cdots,i_n))$, $\forall i, 1 \le i \le p$, and

$\beta' = \phi(/f_1(i_1,i_2,\cdots,i_n),\cdots, f_m(i_1,i_2,\cdots,i_n)/, /l_1 : u_1 : s_1, l_2 : u_2 : s_2, \cdots, l_m : u_m : s_m/)$.

CASE 2:

For the case of a multiple-source data access pattern followed by a single-source data access pattern, we also have to substitute each temporary array in the calculation function.

$$T[i_1,i_2,\cdots,i_n] = \mathcal{F}(\cdots, S[\ f_1(i_1,i_2,\cdots,i_n),\ f_2(i_1,i_2,\cdots,i_n),\cdots,f_m(i_1,i_2,\cdots,i_n)\ ],\cdots) \mid \gamma$$

$$S[i_1,i_2,\cdots,i_m] = Q[\ h_1(i_1,i_2,\cdots,i_m),\ h_2(i_1,i_2,\cdots,i_m),\cdots,h_m(i_1,i_2,\cdots,i_m)\ ] \mid \beta$$

where f_i *and* h_j are the index functions, $\forall\ i, 1 \le i \le m$, *and* $\forall\ j, 1 \le j \le p$, γ and β are segmentation descriptors, and $\beta = \phi(/i_1,i_2,\cdots,i_m/, /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$.

The S in the RHS of the first move pattern can be replaced with Q with respect to the second access pattern. It yields the following access pattern:

$$T[i_1,i_2,\cdots,i_n] = \mathcal{F}(\cdots, Q[\ g_1(i_1,i_2,\cdots,i_n),\cdots,g_p(i_1,i_2,\cdots,i_n)\ ],\cdots) \mid \gamma \wedge \beta'$$

where $g_i(i_1,i_2,\cdots,i_n) = h_i(f_1(i_1,i_2,\cdots,i_n),\cdots, f_m(i_1,i_2,\cdots,i_n))$, $\forall$ i, 1≤i≤p, and

$\beta' = \phi(\ /f_1(i_1,i_2,\cdots,i_n),\cdots,\ f_m(i_1,i_2,\cdots,i_n)/\ , /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$.

CASE 3:

For the case of a single-source access pattern followed by a multiple-source data access pattern.

$$T[i_1,i_2,\cdots,i_n] = S[\ f_1(i_1,i_2,\cdots,i_n),\ f_2(i_1,i_2,\cdots,i_n),\cdots,f_m(i_1,i_2,\cdots,i_n)\ ] \mid \gamma$$

$$S[i_1,i_2,\cdots,i_m] = \mathcal{F}(\ Q_1[h_{1,1}(i_1,i_2,\cdots,i_m),\cdots,h_{1,x_1}(i_1,i_2,\cdots,i_m)]\ ,\cdots,\ Q_2[h_{2,1}(i_1,i_2,\cdots,i_y),\cdots,h_{2,x_2}(i_1,i_2,\cdots,i_m)]\ ,\cdots,\ Q_k[h_{k,1}(i_1,i_2,\cdots,i_y),\cdots,h_{k,x_k}(i_1,i_2,\cdots,i_m)]\ ) \mid \beta$$

where γ and β are segmentation descriptors, and $\beta = \phi(/i_1,i_2,\cdots,i_m/, /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$.

The S in the RHS of the first movement pattern can be replaced with $\mathcal{F}$ with respect to the second access pattern. It yields the following access pattern:

$$T[i_1,i_2,\cdots,i_n] = \mathcal{F}(\ Q_1[g_{1,1}(i_1,i_2,\cdots,i_n),\cdots,g_{1,x_1}(i_1,i_2,\cdots,i_n)]\ ,\cdots,\ Q_2[g_{2,1}(i_1,i_2,\cdots,i_n),\cdots,g_{2,x_2}(i_1,i_2,\cdots,i_n)]\ ,\cdots,\ Q_k[g_{k,1}(i_1,i_2,\cdots,i_n),\cdots,g_{k,x_k}(i_1,i_2,\cdots,i_n)]\ ) \mid \gamma \wedge \beta'$$

where $g_{a,b}(i_1,i_2,\cdots,i_n) = h_{a,b}(f_1(i_1,i_2,\cdots,i_n),\cdots, f_m(i_1,i_2,\cdots,i_n))$, and $\beta' = \phi(\ /f_1(i_1,i_2,\cdots,i_n),\cdots,\ f_m(i_1,i_2,\cdots,i_n)/\ , /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$.

CASE 4:

Suppose we get the following two multiple-sources data access patterns:

$$T[i_1,i_2,\cdots,i_n] = \mathcal{F}1(\cdots, S[\ f_1(i_1,i_2,\cdots,i_n),f_2(i_1,i_2,\cdots,i_n),\cdots,\ f_m(i_1,i_2,\cdots,i_n)]\ ,\cdots) \mid \gamma$$

$S[i_1,i_2,\cdots,i_m]=\mathcal{F}2(\ Q_1[h_{1,1}(i_1,i_2,\cdots,i_m),\cdots,h_{1,x_1}(i_1,i_2,\cdots,i_m)]\ ,\cdots,\ Q_2[h_{2,1}(i_1,i_2,\cdots,i_y),\cdots,h_{2,x_2}(i_1,i_2,\cdots,i_m)]\ ,\cdots,\ Q_k[h_{k,1}(i_1,i_2,\cdots,i_y),\cdots,h_{k,x_k}(i_1,i_2,\cdots,i_m)]\ )\mid\beta$
where γ and β are segmentation descriptors, and $\beta = \phi(/i_1,i_2,\cdots,i_m/\ ,\ /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$.

The S_m in the RHS of the first one can be replaced with Q_p with respect to the second one. It yields the following access pattern:

$T[i_1,i_2,\cdots,i_n]=\ \mathcal{F}1(\cdots,\ \mathcal{F}2(\ Q_1[g_{1,1}(i_1,i_2,\cdots,i_n),\cdots,g_{1,x_1}(i_1,i_2,\cdots,i_n)]\ ,\cdots,\ Q_2[g_{2,1}(i_1,i_2,\cdots,i_n),\cdots,g_{2,x_2}(i_1,i_2,\cdots,i_n)]\ ,\cdots,\ Q_k[g_{k,1}(i_1,i_2,\cdots,i_n),\cdots,g_{k,x_k}(i_1,i_2,\cdots,i_n)]\ )\ ,\cdots)\mid\gamma\wedge\beta'$
where $g_{a,b}(i_1,i_2,\cdots,i_n)=h_{a,b}(f_1(i_1,i_2,\cdots,i_n),\cdots,f_m(i_1,i_2,\cdots,i_n))$, and

$\beta'=\phi(\ /f_1(i_1,i_2,\cdots,i_n),\cdots,\ f_m(i_1,i_2,\cdots,i_n)\ /,\ /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$.

We will then use the example in Figure 1 to show how these data access functions can be composed.

We start with the data access function in the root node (CSHIFT) of the parse tree:

$$B[i,j]=\begin{cases} T4[i+1,j] & \mid\ \phi(/\text{i,j/,/1:3,1:4/})\cdots\ \textbf{(A)}\\ T4[i-3,j] & \mid\ \phi(/\text{i,j/,/4:4,1:4/})\cdots\ \textbf{(B)}\end{cases}$$

We label the two access patterns of the above data access function with (A) and (B). Following is a series of substitutions for this example:

(1) Addition(+): T4[i,j]=$\mathcal{F}$(T2[i,j],T3[i,j]) , $\mathcal{F}$(x,y)=x+y $\cdots$ **(C)**

$\Longrightarrow$

$$B[i,j]=\begin{cases} \mathcal{F}(T2[i+1,j],T3[i+1,j]) & \mid\ \phi(/\text{i,j/,/1:3,1:4/})\cdots\ \textbf{(D)}\\ \mathcal{F}(T2[i-3,j],T3[i-3,j]) & \mid\ \phi(/\text{i,j/,/4:4,1:4/})\cdots\ \textbf{(E)}\end{cases}$$

$\mathcal{F}$(x,y)=x+y

Based on case 3, the access pattern (D) is a composition of (A) and (C), and (E) is a composition of (B) and (C).

(2) TRANSPOSE: using T2[i,j]=T1[j,i] $\cdots$ **(F)**

$\Longrightarrow$ B[i,j] =

$$\begin{cases} \mathcal{F}(T1[j,i+1],T3[i+1,j]) & \mid\ \phi(/\text{i,j/,/1:3,1:4/})\cdots\ \textbf{(G)}\\ \mathcal{F}(T1[j,i-3],T3[i-3,j]) & \mid\ \phi(/\text{i,j/,/4:4,1:4/})\cdots\ \textbf{(H)}\end{cases}$$

$\mathcal{F}$(x,y)=x+y

Based on case 2, the access pattern (G) is a composition of (D) and (F), and (H) is a composition of (E) and (F).

(3) AUGMENT: using

$$T1[i,j]=\begin{cases} A[i,j] & \mid\phi(/\text{i,j/,/1:3,1:3/})\cdots\ \textbf{(I)}\\ 0 & \mid\phi(/\text{i,j/,/4:4,1:4/})\vee\phi(/\text{i,j/,/1:4,4:4/})\cdots\ \textbf{(J)}\end{cases}$$

$\Longrightarrow$ B[i,j] =

$$\begin{cases} \mathcal{F}(A[j,i+1],T3[i+1,j]) & \mid\ \phi(/\text{i,j/,/1:3,1:4/})\wedge\\ & \quad\phi(/\text{j,i+1/,/1:3,1:3/})\cdots\ \textbf{(K)}\\ \mathcal{F}(0,T3[i+1,j]) & \mid\ \phi(/\text{i,j/,/1:3,1:4/})\wedge\\ & \quad\phi(/\text{j,i+1/,/4:4,1:4/})\vee\\ & \quad\phi(/\text{j,i+1/,/1:4,4:4/})\ )\cdots\ \textbf{(L)}\\ \mathcal{F}(A[j,i-3],T3[i-3,j]) & \mid\ \phi(/\text{i,j/,/4:4,1:4/})\wedge\\ & \quad\phi(/\text{j,i-3/,/1:3,1:3/})\cdots\ \textbf{(M)}\\ \mathcal{F}(0,T3[i-3,j]) & \mid\ \phi(/\text{i,j/,/4:4,1:4/})\wedge\\ & \quad\phi(/\text{j,i-3/,/4:4,1:4/})\vee\\ & \quad\phi(/\text{j,i-3/,/1:4,4:4/})\cdots\ \textbf{(N)}\end{cases}$$

$\mathcal{F}$(x,y)=x+y

Based on case 2, the access pattern (K) is a composition of (G) and (I), (L) is a composition of (G) and (J), (M) is a composition of (H) and (I), and (N) is a composition of (H) and (J).

(4) RESHAPE: using T3[i,j]=C[i+4*j-4] $\cdots$ **(O)** $\Longrightarrow$
B[i,j] =

$$\begin{cases} \mathcal{F}(A[j,i+1],C[i+1+4*j-4]) & \mid\ \phi(/\text{i,j/,/1:3,1:4/})\wedge\\ & \quad\phi(/\text{j,i+1/,/1:3,1:3/})\cdots\ \textbf{(K)}\\ \mathcal{F}(0,C[i+1+4*j-4]) & \mid\ \phi(/\text{i,j/,/1:3,1:4/})\wedge\\ & \quad\phi(/\text{j,i+1/,/4:4,1:4/})\vee\\ & \quad\phi(/\text{j,i+1/,/1:4,4:4/})\ )\cdots\ \textbf{(L)}\\ \mathcal{F}(A[j,i-3],C[i-3+4*j-4]) & \mid\ \phi(/\text{i,j/,/4:4,1:4/})\wedge\\ & \quad\phi(/\text{j,i-3/,/1:3,1:3/})\cdots\ \textbf{(M)}\\ \mathcal{F}(0,C[i-3+4*j-4]) & \mid\ \phi(/\text{i,j/,/4:4,1:4/})\wedge\\ & \quad\phi(/\text{j,i-3/,/4:4,1:4/})\vee\\ & \quad\phi(/\text{j,i-3/,/1:4,4:4/})\cdots\ \textbf{(N)}\end{cases}$$

$\mathcal{F}$(x,y)=x+y

Based on case 2, the access pattern (P) is a composition of (K) and (O), (Q) is a composition of (L) and (O), (R) is a composition of (M) and (O), and (S) is a composition of (N) and (O).

Step 4: Code generation

After having derived the synthesized data access function, we can use it to generate a nested loop. For the same example, we use the synthesized data access function obtained in step 3 to generate the code below.

```
FORALL(i=1;i≤4;i++)
  FORALL (j=1;j≤4;j++)
    if( φ(/i,j/,/1:3,1:4/) ∧ φ(/j,i+1/,/1:3,1:3/))
      B[i,j]=A[j,i+1]+C[i+1+4*j-4]
    if(φ(/i,j/,/1:3,1:4/) ∧ (φ(/j,i+1/,/4:4,1:4/) ∨ φ(/j,i+1/,/1:4,4:4/)))
      B[i,j]=C[i+1+4*j-4]
    if(φ(/i,j/,/4:4,1:4/) ∧ φ(/j,i-3/,/1:3,1:3/))
      B[i,j]=A[j,i-3]+C[i-3+4*j-4]
    if(φ(/i,j/,/3:3,0:3/) ∧ (φ(/j,i-3/,/4:4,1:4/) ∨ (φ(/j,i-3/,/1:4,4:4/))
      B[i,j]=C[i-3+4*j-4]
```

4 Optimization

The code generated from step 4 in the previous section does not have any temporary arrays and redundant data movements. It consists of only a single nested loop. However, a number of "if-then" clauses in the code will increase the overhead due to the computing of guarded statements at run-time. We have developed a systematic scheme to reduce this overhead by simplifying the ranges at compilation time instead of run time. Our scheme is done by deriving the bounds of nested loops to cover the index range of a segmentation descriptor of access patterns.

Our scheme works for segmentation descriptors which are linear affined index functions[2]. An index function of a linear segmentation descriptor is of the form $C_0{*}i_j{+}C_1$, where C_0 and C_1 are constants, and i_j is an index variable. Given a simple segmentation descriptor,

[2]The problem is still open if the index function is not linear.

$\phi(/f_1(i_1,i_2, \cdots ,i_n),\cdots , f_m(i_1,i_2, \cdots ,i_n)/ , /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$.

We can get the following by solving diophantine equation[12],

$\phi(/i_1,i_2,\cdots,i_n/ ,/ l'_1{:}u'_1{:}s'_1, l'_2{:}u'_2{:}s'_2,\cdots, l'_n{:}u'_n{:}s'_n/)$, $\forall k$, $1 \le k \le$ n , $l'_k \le i_k \le u'_k$.

When a segmentation descriptor is not simple, we have to take a series of steps to find the matched nested loops. To explain it, let us consider the first segmented descriptor in our example in step 4 of Section 3. Suppose S=ϕ(/i,j/,/1:3,1:4/) $\wedge$ (ϕ(/j,i+1/,/4:4,1:4/) $\vee$ ϕ(/j,i+1/,/1:4,4:4/)), the segmentation descriptor is first converted into a disjunctive normal form[8] $\Rightarrow$

(ϕ(/i,j/,/1:3,1:4/) $\wedge$ ϕ(/j,i+1/,/4:4,1:4/)) $\vee$ (ϕ(/i,j/,/1:3,1:4/) $\wedge$ $\phi[A$(/j,i+1/,/1:4,4:4/))

Then we do the intersection operation:

$\Rightarrow$ ($l_{1,1}{:}u_{1,1}{:}s_{1,1}$, $l_{1,2}{:}u_{1,2}{:}s_{1,2}$, $\cdots$, $l_{1,n}{:}u_{1,n}{:}s_{1,n}$) $\cap$ ($l_{2,1}{:}u_{2,1}{:}s_{2,1}$, $l_{2,2}{:}u_{2,2}{:}s_{2,2}$, $\cdots$, $l_{2,n}{:}u_{2,n}{:}s_{2,n}$)

=($l'_1{:}u'_1{:}s'_1$, $l'_2{:}u'_2{:}s'_2$, $\cdots$, $l'_n{:}u'_n{:}s'_n$) $\forall$ i, 1 $\le$ i $\le$ n, l'_i, u'_i, s'_i derived from $l_{1,i}$, $u_{1,i}$, $s_{1,i}$, $l_{2,i}$, $u_{2,i}$, and $s_{2,i}$.

The above procedure is done by solving the linear diophantine equation to derive the the smallest integer which is in both ($l_{1,i}{:}u_{1,i}{:}s_{1,i}$) and ($l_{2,i}{:}u_{2,i}{:}s_{2,i}$). For example, Assume we have ($l_{1,i}{:}u_{1,i}{:}s_{1,i}$) and ($l_{2,i}{:}u_{2,i}{:}s_{2,i}$) to be (6:120:2) and (5:100:3), respectively. The linear diophantine equation is $6+2*x = 5+3*y$. The solution of this linear diophantine equation are $x = 1-3*t$ and $y = 1-2*t$. Note that both x and y are positive integer $\Rightarrow 1-3*t > 0$ and $1-2*t > 0 \Rightarrow 1/3 > t$ and $1/2 > t \Rightarrow 1/3 > t$. The largest integer solution for $1/3 > t$ is t=0. For $t = 0$ and $x = 1-3*t \Rightarrow$ x=1. We have l'_i=6+2*1=8, u'_i=min($u_{1,i}$,$u_{2,i}$) and s'_i=lcm($s_{1,i}$,$s_{2,i}$). Therefore, the intersection of (6:120:2) and (5:100:3) is (8:100:6).

Applying the procedures, it first yields: ((1:3,1:3) $\cap$ (0:3,4:4)) $\cup$ ((1:3,1:4) $\cap$ (3:3,1:4)), and then after intersection, it yields (1:3,4:4) $\cup$ (3:3,1:4). By using the scheme mentioned above, three other segmentation descriptors of the example can be computed similarly as follows.

ϕ(/i,j/,/1:3,1:4/) $\wedge$ ϕ(/j,i+1/,/1:3,1:3/)
$\Rightarrow$ (1:3,1:4) $\cap$ (0:2,1:3) $\Rightarrow$ (1:2,1:3)

ϕ(/i,j/,/4:4,1:4/) $\wedge$ ϕ(/j,i-3/,/1:3,1:3/)
$\Rightarrow$ (4:4,1:4) $\cap$ (4:6,1:3) $\Rightarrow$ (4:4,1:3)

ϕ(/i,j/,/4:4,1:4/) $\wedge$ (ϕ(/j,i-3/,/4:4,1:4/) $\vee$ (ϕ(/j,i-3/,/1:4,4:4/)
$\Rightarrow$ (ϕ(/i,j/,/4:4,1:4/) $\wedge$ ϕ(/j,i-3/,/4:4,1:4/)) $\vee$ (ϕ(/i,j/,/4:4,1:4/) $\wedge$ ϕ(/j,i-3/,/1:4,4:4/))
$\Rightarrow$ ((4:4,1:4) $\cap$ (4:7,4:4)) $\cup$ ((4:4,1:4) $\cap$ (7:7,1:4))
$\Rightarrow$ ((4:4,1:4) $\cap$ (4:7,4:4)) $\cup$ $\emptyset$
$\Rightarrow$ (4:4,4:4)

Then, the new index range can be used to generate the codes below. Figure 2 also illustrates the composite access pattern after the synthesis and optimization process.

```
FORALL(i=1,i≤2,i++)
  FORALL (j=1,j≤3,j++) B[i,j]=A[j,i+1]+C[i+1+4*j-4]
FORALL(i=1,i≤3,i++)
```

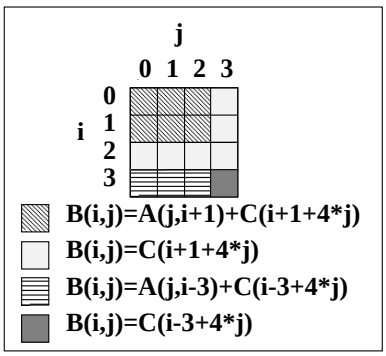

Figure 2: Illustration of Composite Data Access Patterns After Synthesis

```
   FORALL (j=4,j≤4,j++) B[i,j]=C[i+1+4*j]-4
FORALL(i=3,i≤3,i++)
   FORALL (j=1,j≤4,j++) B[i,j]=C[i+1+4*j-4]
FORALL(i=4,i≤4,i++)
   FORALL (j=1,j≤3,j++) B[i,j]=A[j,i-3]+C[i-3+4*j-4]
FORALL(i=4,i≤4,i++)
   FORALL (j=4,j≤4,j++) B[i,j]=C[i-3+4*j-4]
```

5 Synthesis of WHERE Constructs

Based on the statement merge optimization in [6], this synthesis framework can be applied to synthesize the array operations in different array assignment statements. For example, if T is a temporary array, the following two array expressions:

T=TRANSPOSE(CSHIFT(A,1,1))
B=MERGE(T,F,S)
can be transformed to
B=MERGE(TRANSPOSE(CSHIFT(A,1,1)),F,S)

One benefit of statement merge is to eliminate array variables introduced to pass values across statements. In addition, statement merge permits more opportunities for synthesizing consecutive array functions by concatenating statements. The statement merge needs accurate data flow analysis of the reaching and use information and is well treated in [6].

In this paper, we enable the statement merge techniques to include the Fortran 90 WHERE and ELSE-WHERE constructs. This extension is done by translating a WHERE construct into a data access function and then performing the synthesis of the prepared data access function and the data access function of neighboring array statements. This is done with three steps. First, if the body of a WHERE or ELSE-WHERE construct is a compound statement, the compound statement is first synthesized into one data access function. (In the case that the body of a WHERE construct is a single statement, the data access function for the statement is used). After the synthesis of body of WHERE

and ELSE-WHERE constructs, we get a representation which looks as follows:

WHERE (B <> 0)

$$T[i_1,\cdots,i_n] = \begin{cases} \mathcal{G}1[S_1(f_1^1(i_1,\cdots,i_n),\cdots,f_1^{d1}(i_1,\cdots,i_n)],\cdots, \\ \quad S_k[f_k^1(i_1,\cdots,i_n),\cdots,f_k^{dk}(i_1,\cdots,i_n)]) \mid \beta_1 \\ \mathcal{G}2[S_1(h_1^1(i_1,\cdots,i_n),\cdots,h_1^{d1}(i_1,\cdots,i_n)],\cdots, \\ \quad S_k[h_k^1(i_1,\cdots,i_n),\cdots,h_k^{dk}(i_1,\cdots,i_n)]) \mid \beta_2 \end{cases}$$

ElSE-WHERE

$$T[i_1,\cdots,i_n] = \begin{cases} \mathcal{H}1[P_1(f_1^1(i_1,\cdots,i_n),\cdots,f_1^{d1}(i_1,\cdots,i_n)],\cdots, \\ \quad P_k[f_k^1(i_1,\cdots,i_n),\cdots,f_k^{dk}(i_1,\cdots,i_n)]) \mid \gamma_1 \\ \mathcal{H}2[P_1(h_1^1(i_1,\cdots,i_n),\cdots,h_1^{d1}(i_1,\cdots,i_n)],\cdots, \\ \quad P_k[h_k^1(i_1,\cdots,i_n),\cdots,h_k^{dk}(i_1,\cdots,i_n)]) \mid \gamma_2 \end{cases}$$

where f and h are index functions, S_i and P_i are source arrays, B and T are arrays, β_1, β_2, γ_1, and γ_2 are segmented descriptors, and $\mathcal{G}1$, $\mathcal{G}2$, $\mathcal{H}1$, and $\mathcal{H}2$ are calculation functions in the data access function. The body of the WHERE construct now can be either one of the four types of data access function. The example above is a type 4 data access function, which is the most general form. The data access functions of both the body of WHERE and ELSE-WHERE constructs need to have the same target array to be considered in our scheme.

In the second step, the data access functions of body of WHERE and ELSE-WHERE constructs are transformed into a data access function representing the whole WHERE construct as follows:

$$T[i_1,\cdots,i_n] = \begin{cases} \mathcal{F}(\mathcal{G}1(S_1[\cdots],\cdots),\mathcal{H}1(P_1[\cdots],\cdots),B[i_1,\cdots,i_n]) \mid \beta_1 \wedge \gamma_1 \\ \mathcal{F}(\mathcal{G}1(S_1[\cdots],\cdots),\mathcal{H}2(P_1[\cdots],\cdots),B[i_1,\cdots,i_n]) \mid \beta_1 \wedge \gamma_2 \\ \mathcal{F}(\mathcal{G}2(S_1[\cdots],\cdots),\mathcal{H}1(P_1[\cdots],\cdots),B[i_1,\cdots,i_n]) \mid \beta_2 \wedge \gamma_1 \\ \mathcal{F}(\mathcal{G}2(S_1[\cdots],\cdots),\mathcal{H}2(P_1[\cdots],\cdots),B[i_1,\cdots,i_n]) \mid \beta_2 \wedge \gamma_2 \end{cases}$$

$$\mathcal{F}(x,y,z) = \begin{cases} x & \text{if z}\neq 0 \\ y & \text{otherwise.} \end{cases}$$

Finally, the data access function of the WHERE construct can then be synthesized with array statements outside a WHERE statement. An example is shown below to describe this process.

```
REAL A(4,4),T(4,4),B(4,4),C(4,4)
WHERE(E ≠ 0)
        T=B
ELSE WHERE
        T=C
END WHERE
A=TRANSPOSE(CSHIFT(T,1,1))
```

The data access function of the WHERE construct shown in the above code fragment is T[i,j]=$\mathcal{F}$(B[i,j],C[i,j],E[i,j]), and

$$\mathcal{F}(x,y,z) = \begin{cases} x & \text{if z}\neq 0 \\ y & \text{otherwise.} \end{cases}$$

The synthesized data access function of A=TRANSPOSE(CSHIFT(T,1,1)) is

$$A[i,j] = \begin{cases} T[j-3,i] \mid \phi(\text{/j,i/,/4:4,1:4/}) \\ T[j+1,i] \mid \phi(\text{/j,i/,/1:3,1:4/}) \end{cases}$$

We can synthesize these two data access functions in this code fragment, and the final result is

$$A[i,j] = \begin{cases} \mathcal{F}(B[j-3,i],C[j-3,i],E[j-3,i]) \mid \phi(\text{/j,i/,/4:4,1:4/}) \\ \mathcal{F}(B[j+1,i],C[j+1,i],E[j+1,i]) \mid \phi(\text{/j,i/,/1:3,1:4/}) \end{cases}$$

$$\mathcal{F}(x,y,z) = \begin{cases} x & \text{if z}\neq 0 \\ y & \text{otherwise} \end{cases}$$

6 Synthesis Anomaly and Solution

The array synthesis can improve program performance by reducing redundant data movement, temporary storage usage, and loop barrier synchronization overhead. The synthesis scheme, however, may increase the execution time of programs due to the replication of computation while intermediate arrays are eliminated. This occurs when the array considered to be eliminated is replicated by a SPREAD function. (Another case of the performance anomaly is when eliminating temporaries which will be used multiple times. It can be solved similarly.)

For example, in the following code segment:

```
REAL A(N),B(N),C(N),T(N,M),D(N,M)
A=SIN(SQR(B+0.5)+COS(C)) ··· (1)
T=SPREAD(A,DIM=2,NCOPIES=M)+D ··· (2)
```

If we only synthesize array expressions **(1)** and **(2)** individually, the performance will be better than that of synthesizing array expressions **(1)** and **(2)** collectively based on the presented schemes so far. In the first case, the synthesized code is as follows:

```
FORALL(i=1;i≤N;i++)
   A[i]=SIN(SQR(B[i]+0.5)+COS(C[i]))
FORALL(i=1;i≤N;i++)
   FORALL(j=1;j≤M;j++)
      T[i,j]=A[i]+D[i,j]
```

It needs N times of SIN, SQR, and COS arithmetic operations, 2*N+N*M times of floating addition operations, and N+N*M times of assignments. In the second case, the synthesized code is as follows:

```
FORALL(i=1;i≤N;i++)
   FORALL(j=1;j≤M;j++)
      T[i,j]=SIN(SQR(B[i]+0.5)+COS(C[i]))+D[i,j]
```

It needs N*M times of SIN, SQR, and COS arithmetic operations, 3*N*M times of floating addition operations and N*M times of assignments. The example shows that the synthesis may increase the cost of total computation. This is because the SPREAD function in Fortran 90 replicates an array by adding one dimension. It specifies a one-to-many data movement between the source and target arrays.

One way to avoid this performance anomaly is to always divide program segments into two parts and synthesize them separately, when one-to-many operation is used.

In the following, we present an algorithm which can synthesize an important subset of expressions which have one-to-many operations, without producing any performance anomaly.

```
ALGORITHM: Synthesis anomaly prevention
  INPUT: A sequence of array statements to be synthesized.
  (Assume that target array in the LHS of the synthesized
  data access function to be T[i1,i2,···,in])
BEGIN
  /* Step 1: check a one-to-many expression and place
  angle brackets around it */
  IF (there exists no one-to-many array operation)
     Perform array synthesis and code generation as defined
     in previous sections.
     return.
  ELSE
     Perform the modified array synthesis. In the substitution
     process of array synthesis, the expression which substitutes
     the source of a one-to-many array operation is placed within
     a pair of angle brackets.
  /* Step 2: there is a one-to-many operation */
  Remove the pair of angle brackets from the final synthesized
  expression if the expression within the angle brackets does
  not have any arithmetic computation.
  /* Step 3: there is an angle-bracketed expression */
  IF (there exists an array subscript within the angle brackets
        in the RHS of the synthesized data access function
        which is not an admissible prefix of { i1,i2,···,in} )
     Report the statement number, where the expressions inside
     and outside the angle brackets will be synthesized separately.
  ELSE
     (1) For the RHS of the synthesized data access function, choose
     a bracketed subexpression, say φ, which does not contain
     another bracketed subexpression. Assume the loop index
     variables in φ are i1,i2,···,ik.
     (2) Create a temporary array variable TEMP. Modify the
     synthesized data access function by replacing φ with
     TEMP and moving TEMP=φ to the outer loop of loop index
     ik.
     (3) Repeat (1) and (2) until all of the bracketed subexpressions
     in the RHS of the synthesized data access function are replaced
     with created temporary variables, and the corresponding
     subexpressions are moved to the outer loop.
END
```

This algorithm is divided into three steps. In the first step, it checks if any SPREAD operation exists. If a SPREAD function does not exist, array operation synthesis is performed based on our earlier scheme. If there is a one-to-many operation, then during the substitution process of the array synthesis, the expression which substitutes the source of a one-to-many array operation is placed within a pair of angle brackets. In our example, we get the following data access pattern:

T[i,j]={SIN(SQR(B[i]+0.5)+COS(C[i]))}+D[i,j]

In general, there may be multiple bracketed subexpressions, where each one represents a one-to-many data movement between the source and target arrays.

In the second step, we examine the final data access function after synthesis. The algorithm removes the angle brackets from the final synthesized expression if the expression within a pair of angle brackets does not have any arithmetic computation. The reason is that if there is no computation, there is no replicated computation produced.

The final stage is the code generation stage. In the code generation stage, the duplicated computation can be avoided by moving the bracketed subexpression to an outer loop. If the array subscript within a pair of angle brackets is an *admissible prefix* (defined below) of the index variables of the target array, we can always move the bracketed subexpression to an outer loop. Otherwise, we will report that the statements should be split into two parts for synthesis.

Definition 5 Assume S = $\{a_1,a_2,\cdots,a_n\}$ is a sequence of ordered set. AP = $\{b_1,b_2,\cdots,b_k\}$ is an **admissible prefix** of S if k $\leq$ n and $b_1,b_2,\cdots,b_k$ is a permutation of $a_1,a_2,\cdots,a_k$.

We show a complete example below.

```
REAL A(N,M),B(N,M),C(N),D(N,M,L),G(N),
       T(N,M,L),E(N,M)
G=COS(C)
A=SIN(SQR(B+0.5)+SPREAD(G,DIM=2,NCOPIES=M))
T=SPREAD(A,DIM=3,NCOPIES=L)+D+
    SPREAD(E,DIM=3,NCOPIES=L )
```

After step (1), we get the synthesized code as follows:

```
T[i,j,k]={SIN(SQR(B[i,j]+0.5)+{COS(C[i])})}+D[i,j,k]+{E[i,j]}
```

Step 2 eliminates the pair of angle brackets of E in this case and gets

```
T[i,j,k]={SIN(SQR(B[i,j]+0.5)+{COS(C[i])})}+D[i,j,k]+E[i,j].
```

In the step 3, we first choose {COS(C[i])} and move it to the outer loop of loop index i and get {SIN(SQR(B[i,j]+0.5)+Temp1)}. We then move {SIN(SQR(B[i,j]+0.5)+Temp1)} to the loop of loop index j. The final synthesized code is shown below.

```
FORALL(i=1;i≤N;i++)
   Temp1=COS(C[i])
   FORALL(j=1;j≤M;j++)
      Temp2=SIN(SQR(B[i,j]+0.5)+Temp1)
      FORALL(k=1;k≤L;k++)
         T[i,j,k]=Temp2+D[i,j,k]+E[i,j]
```

7 Experiments

Seven test suites shown in the appendix A are used to evaluate the effectiveness of our synthesis scheme in improving the execution time performance of Fortran 90 programs with array operations. Both of the base and synthesized codes in these experiments are hand-translated to C codes, which are then compiled by the native C compiler on a Sequent multiprocessor machine. The base version is based on a straightforward compilation by translating each Fortran 90 array intrinsic function into a (parallel) nested loop and uses a temporary array to pass intermediate results. (In addition, loop fusion and statement merge are done in the base version so that there is only one single loop generated for expression like A(:) = B(:) + C(:) + D(:).) The synthesized codes are obtained from hand-translations based on our proposed algorithms. The execution time shown in Table 1 is obtained by running each program fragment on an 8-node Sequent shared-memory multiprocessor machine. The speedup is the performance improvement of the synthesis scheme over the base version with the same number of processors. The first three program segments are created using the combinations of SPREAD, RESHAPE, EOSHIFT, MERGE,

Time (seconds)	p=1	p=2	p=4	p=8
(1) (Example) base code	47.41	24.2	12.9	6.35
(1) Synthesized Code	17.08	8.8	4.7	2.37
(1) Improvement	2.77	2.75	2.74	2.67
(2) (Where) base code	26.56	13.35	6.90	3.61
(2) Synthesized code	17.01	8.55	4.31	2.24
(2) Improvement	1.56	1.56	1.60	1.61
(3) (EMR) Base code	36.97	19.41	10.06	5.23
(3) Synthesized code	16.76	8.62	4.40	2.28
(3) Improvement	2.20	2.25	2.28	2.29
(4) (PURDUE-Set Prob9)	193.11	101.37	51.44	26.5
(4) Synthesized code	104.76	52.61	26.60	13.4
(4) Improvement	1.84	1.92	1.93	1.97
(5) (Magnetic E.) base code	95.3	54.01	27.21	13.76
(5) Synthesized code	76.6	44.55	22.35	11.20
(5) Improvement	1.24	1.21	1.21	1.22
(6) (Sandia Wave)	82.88	44.53	25.24	18.99
(6) Synthesized code	49.30	25.06	12.92	6.44
(6) Improvement	1.68	1.78	1.95	2.95
(7) (Templates from Solver)	197.03	98.29	49.03	24.53
(7) Synthesized code	72.96	37.16	18.85	9.59
(7) Improvement	2.70	2.64	2.60	2.55

Table 1: Performance of test suites on a Sequent with 8 nodes. The performance improvement is measured with the execution time of the the base version over the synthesis version with equal number of processors.

and CSHIFT functions as well as WHERE and ELSEWHERE constructs. They are used to demonstrate the ability of our scheme in composing these operations and constructs and to show the performance effects. The first test case is the example that we use in Section 3. The second example includes WHERE and ELSEWHERE constructs and SPREAD function. The third example is to synthesize EOSHIFT, MERGE, and RESHAPE functions. The measured speedups for these three suites are in the range between 1.56 in suite 2 and 2.77 in suite 1. The optimizations remain effective as the number of processors increases.

The last four program fragments are from real application codes. The dimension size of the data grid is 256. The execution times of suites 4, 5 and 6 are measured by running the program fragments 20 times. Test suite 4 is the inner most loop in the Problem 9 of Purdue Set[9][10]. Most of the programs in the Purdue Set were extracted from large real applications. Problem 9 is to perform a logarithmic transformation $d_i = log(1 + d_i)$ for a set of data $d_i, i = 1, \cdots, N$ and compute first four Fourier moments $F_j = \Sigma_{i=1}^{N} d_i * cos(Pi * j/(N + 1))$. Our optimizations improve performance by nearly two times. The code fragment in the fifth suite is from the APULSE routine in the electromagnetic scattering problem[9]. The performance is improved by more than 20 percent regardless of the number of processors. In addition, we employ solution to the performance anomaly scheme presented earlier in this test case. The code fragment in the sixth suite is modified from the Sandia Wave application. The performance is improved by nearly three times on eight processors. The last code

Programs	(A)	(B)	(C)	(D)
(1) (Example) base code	$6N^2$	$5N^2$	5	$7N^2$
(1) Synthesized Code	$3N^2$	N^2	1	$3N^2$
(2) (Where) base code	$4N^2$	$3N^2$	3	$4N^2 + N$
(2) Synthesized code	$2N^2$	N^2	1	$2N^2 + N$
(3) (EMR) Base code	$6N^2 + N$	$4N^2 + N$	4	$8N^2$
(3) Synthesized code	N^2	N^2	3	$5N^2$
(4) (PURDUE-Set Prob9)	$21N^2$	$11N^2$	11	$12N^2$
(4) Synthesized code	$10N^2$	$N^2 + 8N$	1	$6N^2$
(5) (Magnetic E.) base code	$5N^2$	$3N^2$	4	$3N^2 + 3N$
(5) Synthesized code	$3N^2$	N^2	1	$N^2 + 3N$
(6) (Sandia Wave code)	$16N^2$	$5N^2$	5	$10N^2$
(6) Synthesized code	$12N^2$	N^2	3	$6N^2$
(7) (Templates from Solver)	$7/4 * \Omega$	$5/4 * \Omega$	5	$5N^2 + 32N$
(7) Synthesized code	$3/4 * \Omega$	$1/4 * \Omega$	1	$N^2 + 20N$

$\Omega = \Sigma_{i=1}^{n-1}(i^4 + 2i^3 + i^2)$

(A) : Number of loads (B) : Number of stores
(C) : Number of loop barrier synchronizations
(D) : Array storages used

Table 2: Number of loads, stores, barrier synchronizations, and array storages used in our test suites

segment is from the Linear Equation Solver in Brainberd's book[2]. This program shows up to a 2.7 folds improvement.

Table 2 lists the number of store, load, barrier synchronization, and array storage used in the base and synthesized version respectively. N is the size of the array in each dimension. The column for array in the table lists the number of words totally used in the test suites. In addition, we only count the number of load and store for array elements in the table, as they are the main differences in the synthesis schemes. The number of data movements, are significantly reduced in all cases, which are the main reasons for the performance improvement. In addition, the number of loop barrier and array storages used are all reduced, which, too, contributes to performance improvements.

Finally, we compare our synthesis scheme with the scheme of Guibas[4] and Budd[3], which synthesizes APL array operators by merging their steppers (including the mapping of axis, start, and stride). For those array functions which can be expressed by steppers, the work in Guibas and Budd are as effective as our proposed synthesis scheme and require no intermediate array temporaries and separate loops. These functions include elemental functions, TRANSPOSE, SPREAD, and some very special cases of RESHAPE. However, for the constructs, such as CSHIFT, EOSHIFT, MERGE, and WHERE, which cannot be expressed by simple mapping defined in a stepper, our synthesis scheme shows performance and storage advantages. In the seven test cases, the stepper scheme fails to synthesize the programs in the test suite 1, 2, 3, 4, 6, and produces the same programs as our base version. The stepper mechanism only successfully synthesize the program of the test suite 7, and produces the same synthesized codes as our proposed scheme. In the test suite 5, the stepper successfully synthesizes the first 6 lines of

the program with SPREAD and elemental array operations, but fails to synthesize the rest of the program. However, there is no scheme in the work of Guibas and Budd, to avoid performance anomaly with the SPREAD operations. Performance anomaly does happen with the test suite 5. Our solution in Section 6 helps our scheme to produce optimized codes by avoiding performance anomaly problems.

8 Conclusion

Array operation synthesis transforms several array operations into a data access function. Thus, the synthesis can improve performance by reducing redundant data movement, temporary storage usage, and parallel loop synchronization overhead. In this paper, we have presented a new function composition approach to synthesize array operations. Our scheme can handle not only data accesses between arrays of different numbers of dimensions but also masked array expressions, multiple sources array operations, and segmented array functions, which can not be done collectively in previous research work. As a result, we can handle compositions of RESHAPE, SPREAD, EOSHIFT, TRANSPOSE, CSHIFT, and MERGE functions, array expressions, WHERE constructs, and ELSE-WHERE constructs in Fortran 90. Preliminary experimental results from an 8-node Sequent machine show that our synthesis scheme can significantly improve the performance of all measured Fortran 90 code fragments regardless of the number of processors. The measured speedups from real applications vary between 1.21 and 2.95.

References

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook complete ANSI/ISO reference*, Intertext Publications McGraw-Hill Book Company, 1992.

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

[3] T. A. Budd. *An APL Compiler for a Vector Processor*, ACM Transactions on Programming Languages and Systems, V6, No 3, pp. 297–313, July 1984.

[4] L. J. Guibas and D. K. Wyatt. *Compilation and Delayed Evaluation in APL*, Proceeding of the Fifth Annual ACM Symp. on Principles of Programming Languages, pp. 1-8, 1978.

[5] Dz-ching Ju, Chuan-lin Wu, and Paul Carini, *The Synthesis of Array Functions and its Use in Parallel Computation*, Proceeding of International Conference on Parallel Processing, pp. 293–296, 1992.

[6] Dz-ching Ju. *The Optimization and Parallelization of Array Language Programs*, Ph.D dissertation, University of Texas at Austin, Aug 1992.

[7] Jenq Kuen Lee and D. Gannon. *Object-Oriented Parallel Programming: Experiments and Results, Proceedings of Supercomputing '91*, New Mexico, November, 1991.

[8] C. L. Liu. *Elements of Discrete mathematics*, Second Edition, 1985.

[9] A. Mohamer, G. Fox, G. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, N. and Yeh. *Applications Benchmark Set for Fortran-D and High Performance Fortran*, Technical Report SCCS327, NPAC, Syracuse University.

[10] J.R. Rice, J. Jing. *Problems to Test Parallel and Vector Languages*, Purdue University Tech. Rep. CSD-TR-1016, 1990.

[11] High Performance Fortran Forum, High Performance Fortran Language Specification, Technical Report CRPC-TR92225, Rice University, Houston, TX, 1993.

[12] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*, Addison-Wesley Publishing Company, May 1990.

A Code Fragments for Experiments

1.
```
B=CSHIFT(TRANSPOSE(AUGMENT(A)+RESHAPE(C),1,1))
```

2.
```
T1 = SPREAD(S);
WHERE (T !=0)
  T2 = T1;
ELSEWHERE
  T2 = SIN(T1) +0.5;
B=TRANSPOSE(T2);
```

3.
```
C=EOSHIFT(MERGE(RESHAPE(S,/N,N/),A+B,T))
```

4.
```
  RIP=EOSHIFT(U,DIM=1,SHIFT=1)
  RIN=EOSHIFT(U,DIM=1,SHIFT=-1)
  T=U+RIP+RIN+EOSHIFT(U,DIM=2,SHIFT=1)+
*    EOSHIFT(U,DIM=2,SHIFT=-1)+
*    EOSHIFT(RIP,DIM=2,SHIFT=1)+
*    EOSHIFT(RIP,DIM=2,SHIFT=-1)+
*    EOSHIFT(RIN,DIM=2,SHIFT=1)+
*    EOSHIFT(RIN,DIM=2,SHIFT=-1)
  T=T/PATTERN
  WHERE (ABS(U).GT.0.001)
    ERRM=ABS((U-T)/U)
  ELSEWHERE
    ERRM=ABS((U-T)/0.001)
  ENDWHERE
```

5.
```
KIA=SPREAD(KII,DIM=2,NCOPIES=KM)
KJA=SPREAD(KJJ,DIM=1,NCOPIES=KN)
XM=(KMM+0.5)*HW1/KM*SGN(START1)
XN=(KNN+0.5)*HW2/KN*SGN(START2)
XMA=SPREAD(XM,DIM=1,NCOPIES=KI)
XNA=SPREAD(XN,DIM=2,NCOPIES=KJ)
WHERE(KIA.EQ.1)
YI=1./SQRT(HW1)
ELSEWHERE
YI=SQRT(2./HW1)*COS(3.141592654*(KIA-1)*XMA/HW1)
END WHERE
WHERE(KJA.EQ.1)
YJ=1./SQRT(HW2)
ELSEWHERE
YJ=SQRT(2./HW2)*COS(3.141592654*(KJA-1)*XNA/HW2)
END WHERE
```

6.
```
FXP = CSHIFT(F1,1,+1)
FXM = CSHIFT(F1,1,-1)
FYP = CSHIFT(F1,2,+1)
FYM = CSHIFT(F1,2,-1)
FDERIV = ZXP * (FXP - F1) + ZXM * (FXM - F1) +
       ZYP * (FYP - F1) + ZYM * (FYM - F1)
```

7.
```
INTEGER :: N, K
REAL, DIMENSION (:), INTENT (IN) :: B
REAL, DIMENSION (SIZE (B), SIZE(B) + 1) :: M
N = SIZE(B)
DO K = 1, N-1
M(K+1:N,K+1:N+1)=M(K+1:N,K+1:N+1)-
            SPREAD (M(K,K+1:N+1),1,N-K)*
            SPREAD (M(K+1:N,K),2,N-K+1)
ENDDO
```

B Data access functions

In the appendix, we want to present examples of how to represent an intrinsic array function with our data access function. We will mainly consider the intrinsic functions defined in Fortran 90 and mention some array functions in other program languages.

TRANSPOSE, type 1, Fortran 90 standard intrinsic function : the matrix transposition of the argument array

For example, B=TRANSPOSE(A), the corresponding data access function is

$\Rightarrow$ B[i,j] = A[j,i]

CSHIFT, type 3, Fortran 90 standard intrinsic function : circular shift of the elements of the argument array

(1)R=CSHIFT(M,SHIFT=l,DIM=k)

$$If\ \ l > 0 \Rightarrow$$

$$R[i_1,\cdots,i_n] = \begin{cases} M[i_1,\cdots,i_k+l,\cdots,i_n] & | \ \phi(/i_1,\cdots,i_k,\cdots,i_n/, \\ & \quad /*,*,\cdots,1:l,\cdots,*,*/) \\ M[i_1,\cdots,i_k-s+l,\cdots,i_n] & | \ \phi(/i_1,\cdots,i_k,\cdots,i_n/, \\ & \quad /*,*,\cdots,l+1:s,\cdots,*,*/) \end{cases}$$

$$If\ \ l < 0 \Rightarrow$$

$$R[i_1,\cdots,i_n] = \begin{cases} M[i_1,\cdots,i_k+l,\cdots,i_n] & | \ \phi(/i_1,\cdots,i_k,\cdots,i_n/, \\ & \quad /*,*,\cdots,1:|l|,\cdots,*,*/) \\ M[i_1,\cdots,i_k+s+l,\cdots,i_n] & | \ \phi(/i_1,\cdots,i_k,\cdots,i_n/, \\ & \quad /*,*,\cdots,|l|+1:s,\cdots,*,*/) \end{cases}$$

"s" is the size of dimension k of array M.

(2)R=CSHIFT(M,SHIFT=/-1,1,0/,DIM=2)

In the case, the shifted amount in each row is different.

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow R = \begin{bmatrix} 3 & 1 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix}$$

$$\Rightarrow R[i,j] = \begin{cases} M[i,j+2] & | \ \phi(/i,j/,/1:1,1:1/) \\ M[i,j-1] & | \ \phi(/i,j/,/1:1,2:3/) \\ M[i,j+1] & | \ \phi(/i,j/,/2:2,1:2/) \\ M[i,j-2] & | \ \phi(/i,j/,/2:2,3:3/) \\ M[i,j] & | \ \phi(/i,j/,/3:3,1:3/) \end{cases}$$

EOSHIFT, type 3, Fortran 90 standard intrinsic function : end-off shift of the elements of the argument array

(1)R=EOSHIFT(M,SHIFT=l,BOUNDARY="A",DIM=2)

$$If\ \ l > 0 \Rightarrow$$

$$R[i_1,\cdots,i_n] = \begin{cases} M[i_1,\cdots,i_k+l,\cdots,i_n] & | \ \phi(/i_1,\cdots,i_k,\cdots,i_n/, \\ & \quad /*,*,\cdots,1:l,\cdots,*,*/) \\ A & | \ \phi(/i_1,\cdots,i_k,\cdots,i_n/, \\ & \quad /*,*,\cdots,l+1:s,\cdots,*,*/) \end{cases}$$

$$If\ \ l < 0 \Rightarrow$$

$$R[i_1,\cdots,i_n] = \begin{cases} A & | \ \phi(/i_1,\cdots,i_k,\cdots,i_n/, \\ & \quad /*,*,\cdots,1:|l|,\cdots,*,*/) \\ M[i_1,\cdots,i_k+s+l,\cdots,i_n] & | \ \phi(/i_1,\cdots,i_k,\cdots,i_n/, \\ & \quad /*,*,\cdots,|l|+1:s,\cdots,*,*/) \end{cases}$$

"s" is the size of dimension k of array M.

(2)R=EOSHIFT(M,SHIFT=/-1,1,0/,BOUNDARY=/"A","B","C"/,DIM=2)

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow R = \begin{bmatrix} A & 1 & 2 \\ 5 & 6 & B \\ 7 & 8 & 9 \end{bmatrix}$$

$$\Rightarrow R[i,j] = \begin{cases} A & | \ \phi(/i,j/,/1:1,1:1/) \\ M[i,j-1] & | \ \phi(/i,j/,/1:1,2:3/) \\ M[i,j+1] & | \ \phi(/i,j/,/2:2,1:2/) \\ B & | \ \phi(/i,j/,/2:2,3:3/) \\ M[i,j] & | \ \phi(/i,j/,/3:3,1:3/) \end{cases}$$

MERGE, type 2, Fortran 90 standard intrinsic function : combines two (conformable) arrays under the control of a mask

For example, R=MERGE(T,F,S), the corresponding data access function is

$\Rightarrow$ R[$i_1,\cdots,i_n$]=$\mathcal{F}$(T[$i_1,\cdots,i_n$],F[$i_1,\cdots,i_n$],S[$i_1,\cdots,i_n$])

$$\mathcal{F}(x,y,z) = \begin{cases} x & \text{if } z > 0 \\ y & \text{if } z = 0 \end{cases}$$

SPREAD, type 1, Fortran 90 standard intrinsic function : replicates an array by adding one dimension

For R=SPREAD(SOURCE=S,DIM=d,NCOPIES=n), we have to consider the following two cases:

(1) S is a scalar variable:

$$R[i] = S$$

(2) S is an array:

$$R[i_1,i_2,\cdots,i_d,\cdots,i_n] = S[i_1,\cdots,i_{d-1},i_{d+1},\cdots,i_n]$$

RESHAPE, type 2, Fortran 90 standard intrinsic function : constructs an array of a specified shape from the elements of a given array

(1)For R=RESHAPE(SOURCE=M,SHAPE=S,ORDER=O), where M is a one-dimensional array,

$$R[i_1,i_2,\cdots,i_n] = M[f(1)+f(2)+f(3)+\cdots+f(n)]$$

$$f(k) = \begin{cases} i_k & if\ O[k]=1 \\ (i_k-1)*(S[O[l-1]]*S[O[l-2]]*\cdots*S[O[1]] & if\ O[k]=1\ l\neq 1 \end{cases}$$

(2)If the optional argument PAD appears : For example, T=RESHAPE(A,/4,4/,b,/2,1/).

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} a & b & c & d & e & f & g & h & i & j \end{bmatrix}$$

$$\Rightarrow T = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & a & b \\ c & d & e & f \\ g & h & i & j \end{bmatrix}$$

The data access function is as follows:

$$T[i,j] = \begin{cases} A[4*(i-1)+j] & | \ \phi(/i,j/,/1:1,*/) \vee \phi(/i,j/,/2:2,1:2/ \\ B[(i-2)+(j-2)] & | \ \phi(/i,j/,/2:2,3:4/) \\ B[4*(i-3)+(j+2)] & | \ \phi(/i,j/,/3:4,*/) \end{cases}$$

The following are not Fortran 90 standard intrinsic functions.

REVERSE, type 1, creates an array with elements reversed along a specified axis (There exists such an array operator in APL.)

For example, R=REVERSE(M,D), the corresponding data movement function is

$$\Rightarrow R[i_1,i_2,\cdots,i_D,\cdots,i_n] = M[i_1,i_2,,\cdots,s-i_D+1,\cdots,i_n]$$

"s" is the size of dimension D of array M.

CATENATION, type 3, Catenates two array along a specified axis (There exists such an array operator in APL.)

For example, R=CATENATION(L,R,D), the corresponding data access function is

$\Rightarrow$ $R[i_1,i_2,\cdots,i_D,\cdots,i_n]$ =

$$\begin{cases} L[i_1,i_2,\cdots,i_D,\cdots,i_n] & | \ \phi\ (/i_1,i_2,\cdots,i_D,\cdots,i_n/, \\ & \quad /*,\cdots,1{:}s_1,\cdots,*/) \\ R[i_1,i_2,\cdots,i_D-s_1,\cdots,i_n] & | \ \phi\ (/i_1,i_2,\cdots,i_D,\cdots,i_n/, \\ & \quad /*,\cdots,s_1{+}1{:}s_1{+}s_2,\cdots,*/) \end{cases}$$

"s" is the size of dimension D of array L.

Summary of Fortran 90 array functions

This framework can synthesize Fortran 90 element-wise array expressions, MERGE, SPREAD, RESHAPE, CSHIFT, EOSHIFT, and TRANSPOSE functions, and WHERE constructs. Not all the array functions in Fortran 90 can be represented with our data access functions. For example, the PACK and UNPACK functions can not be represented in this framework. Array reduction functions and array location functions are not element-wise array data access functions and require future enhancements to be incorporated into this framework.