A Function-Composition Approach to Synthesize Fortran 90 Array Operations

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

Dz-Ching R. 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 array operations or array expressions into a composite access function of the source arrays. Our scheme is based on the composition of access functions, which is analogous to a composition of mathematic functions. Our new scheme can handle not only data movements of arrays with different numbers of dimensions and with multiple-clause array operations but also masked array expressions and multiple-source array operations. As a result, our proposed scheme is the first synthesis scheme which can collectively synthesize Fortran 90 RESHAPE, EOSHIFT, MERGE, array reduction operations, and WHERE constructs. In addition, we also discuss the case that the synthesis scheme may result in a performance anomaly due to the presence of common subexpressions and one-to-many array operations. A solution is proposed to avoid such a performance anomaly. Experimental results show speedups from 1.21 to 2.95 over the base code for code fragments from real applications on a Sequent multiprocessor machine, and also show comparable performance improvements on an 8-node SGI Power Challenge by incorporating our proposed optimizations.

1 Introduction

An increasing number of programming languages, such as Fortran 90 [3], HPF (High Performance Fortran) [19], APL, and possibly C++ with built-in array classes [20], provide a rich set of intrinsic array functions and array constructs. These intrinsic array functions and array expressions operate on the elements of multi-dimensional array objects concurrently without requiring iterative statements. These array operations provide a rich source of data parallelism. 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. Synthesis of multiple consecutive array functions or array expressions can compose several

*The correspondence author's e-mail address is jklee@cs.nthu.edu.tw. This paper is submitted to Journal of Parallel and Distributed Computing, and this is the revision version.

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",1)) + RESHAPE(C,/4,4/)),1,1)

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

In this paper, we propose a new array operation synthesis scheme to find the synthesized function $\mathcal{F}'$. Our scheme is based on the composition of mathematic functions. It starts with deriving a mathematic access function for each array expression or intrinsic array function. We then use a function composition approach to compose multiple access functions. This composition is semantically equivalent to the synthesis of array functions. Our scheme can handle arrays with different numbers of dimensions, masked array expressions, multiple-source array operations, array location operations, array reduction operations, and multiple-clause array functions[1]. Although the proposed scheme is applicable to any programming language with array expression syntax or intrinsic array functions, we are particularly interested in optimizing Fortran 90 array language programs as it has become a de facto data parallel language in the scientific and engineering communities. As a result, we can handle compositions of extensive Fortran 90 array constructs, such as RESHAPE, SPREAD, EOSHIFT, TRANSPOSE, CSHIFT, and MERGE functions, array sections, array reduction functions, WHERE and ELSE-WHERE constructs. None of the existing synthesis schemes can synthesize such an extensive set of array operations.

In addition, in the absence of common subexpressions and one-to-many array operations, our array synthesis scheme always produces codes which are more efficient than the base codes in the following respect. The synthesis scheme never increases the total number of stores and loads and the computation time. In the presence of common subexpressions and one-to-many array operations, the number of loads and the computation time may increase. We call this phenomenon synthesis anomaly. A solution is proposed to avoid synthesis anomaly. In this paper, we assume the underlying architecture a flat shared memory parallel machine. Our experiments are done on two platforms: a Sequent multiprocessor and a SGI Power Challenge machine. Experimental results from a Sequent multiprocessor machine show speedups ranging from 1.21 to 2.95 for code fragments from real applications by incorporating the proposed optimizations. Our scheme is equally effective when applied on a SGI Power Challenge environment. These results show that this synthesis scheme is an effective technique in improving the performance of programs with array operations.

The remainder of the paper is organized as follows. Section 2 summarizes the related work. Section 3 illustrates array operations and data access functions, which are the foundation to our array operation synthesis. Section 4 details our scheme in synthesizing array operations. Section 5 discusses an optimization of index functions generated by our synthesis scheme. Section 6 discusses a synthesis anomaly behavior and proposes a solution to the problem. Section 7 presents additional synthesis techniques to synthesize the array reduction (location) operations and WHERE constructs in Fortran 90. Section 8 analyzes the complexity of our synthesis scheme. Section 9 shows our experimental results, and Section 10 concludes this paper.

[1] We present the details of array operations in Section 3.

2 Previous Work

Much of earlier work on optimizing array language programs can be traced back to the research done for compiling APL programs. Guibas and Wyatt [14] first proposed a universal selector representation called a stepper to encode the way in which elements of the current function take part 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. A number of array optimization schemes have since been developed based on the stepper representation. Budd [6] presented several variant data request representations: arithmetic progression vector, column vector, and offset vector, which aimed at vector machines. Ju, Wu, and Carini [15] proposed a scheme which is derived from the stepper representation, but they considered a Concatenate function which was not treated in [14]. The method in [15], however, did not support the data accesses between arrays of different dimensional space, masked array expressions, and array operations of multiple sources. In the stepper mechanism, 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, is insufficient to handle masked array expressions and array operations of multiple sources. As a consequence, these stepper-based mechanisms fail to synthesize the Fortran 90 constructs, such as MERGE, RESHAPE function, and WHERE construct.

Balasundaram and Kennedy proposed the use of simple sections and data access descriptors to approximate an array reference pattern in a loop [4]. Their method was used to detect task parallelism between program loops, but they are insufficient to represent the access patterns of many array constructs, such as SPREAD, MERGE, and WHERE. Chatterjee et al. developed a size and access inference scheme to step up the grain size of computation and reduces synchronizations for executing data parallel programs on multiprocessor machines[7] [9]. They build a global computation graph for a program and partition the graph into clusters, where the generation and consumption patterns of data vectors are derived. In contrast, our approach synthesizes adjacent array operations in expression trees based on the data access functions of these array operations. This allows more efficient compilation time and space usage. Although their work has a goal similar to ours, we handle more Fortran 90 constructs, such as EOSHIFT, CSHIFT, and WHERE. Our study reports similar speedups to theirs on corresponding numbers of processors, though different benchmarks and systems were used.

Waters [25] pointed out the importance of array-style programming and optimization for array operations. The focus of his work was to list a number restrictions in user programs for efficient transformations of *Series* (unbounded vectors) expressions and communicate these restrictions to the users through static verification by a compiler. The result of generating pipelined code for array optimization in his work is similar to our synthesis scheme wants to accomplish. His work, however, does not provide solutions to compose array operations with a mismatched ordering between a source and a target, such as CSHIFT, EOSHIFT, SPREAD, and RESHAPE, and to handle WHERE constructs due to a straight-line computation constraint. Walinsky and Banerjee[26] described the development of a compiler for a functional language, FP, on CM-2 machines. Their work identified a particular class of data rearrangement functions, called routing functions, which are amenable to compiler analysis and optimization. Routing functions can be optimized with "definition propagation". Dead-code elimination and index renaming are used to help gain more opportunities for doing definition propagations. The idea of definition propagations is similar to our array operation synthesis. However, the routing functions handled in their work are quite limited. Their method does not handle array operations with segmented data movements in different index ranges. As a result, definition propagation in [26] does not handle operations such as CSHIFT, EOSHIFT, reduction operations, and WHERE constructs.

Our main framework here is to compose consecutive array operations or array expressions in Fortran 90 into a composite function of the source arrays to improve program performances. In addition to the main synthesis framework, two auxiliary techniques are needed to solidify the performance gains during the synthesis process. As we will see later in this paper, the two auxiliary techniques are for array descriptor optimizations and synthesis anomaly solutions. Work related to those two techniques are given below. A number of previous work are related to array descriptor optimization. Similar approaches were used in code generation for array assignments [12, 24, 10, 23], in HPF, but they were aimed at different purposes. Havlak and Kennedy [17] also described the techniques to combine and merge array descriptors. Sips et al.[23] developed V-cal calculus for SPMD code generations and various optimizations. One of their optimizations is the compile-time conversion of these sets to "closed form" index sets, which is important to reduce run-time overhead. Their concept is similar to our optimization for array descriptors, but we extend the code generation techniques to synthesized codes with coupled index functions.

In the work related to synthesis anomaly handling, Chatterjee, Gilbert, and Schreiber [8] discussed how to align arrays on distributed-memory machines to reduce communication costs. Their method identifies situations in which replicated alignment is either required by the program itself (via SPREAD operations) or can be used to improve performance. In the presence of a SPREAD operation within a do loop, a broadcast will occur in every iteration if the input array of the SPREAD operation is not replicated, while a single broadcast will occur (at loop entry) if it is replicated. The solution in their work is to apply a network flow algorithm. It will first construct an align distribution graph (ADG) and then apply the network flow algorithm to find the optimal replication labeling. Their solution should be applicable to our problem only that their work tries to reduce communication, while our work tries to reduce the computation. In addition, we incorporate loop interchanges for further synthesis (Section 6.2 of our paper) opportunities. Knobe and Dally[18] described a shape-based abstraction for compiler optimizations. They also suggested "invariant code motion" which is a transformation that removes a loop index from the shape of an operation when the results of all iterations are identical. Their idea in performing loop transformations to gain performances is similar to our work in solving synthesis anomaly.

A preliminary result of our research was published in [13]. We have significantly strengthened the study in this paper through the following respects. The optimization of combining predicates of testing index ranges and loop bounds is further formalized and extended to cover additional cases, such as coupled index functions. In addition, our synthesis scheme is extended to deal with array reduction functions, an important set of Fortran 90 intrinsic functions. Finally, we provide experimental results at an 8-node SGI Power Challenge machine in addition to the earlier results from a Sequent multiprocessor machine to demonstrate that our scheme is an effective scheme for array operation optimizations on different platforms.

3 Data Access Functions of Array Operations

Our scheme starts with the derivation of a mathematic function for each array expression or intrinsic array function to characterize the access pattern of source array elements. We then use a function composition approach to compose these functions. In this section, we will first introduce the mathematic function describing an array access pattern, which is called "data access function" in our work. We will then describe how each array intrinsic function and array operation can be mapped to a "data access function".

We first show an example of a data access function for the CSHIFT operation in Fortran 90, and the general definition of data access functions is given later.

Example 1 Let that A and B be 4 by 4 matrices, the data access function of B=CSHIFT(A,1,1) is

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

The above equation specifies that A[i-3,j] is moved to B[i,j] when (i, j) is in the range of (4:4,1:4), and A[i+1,j] is moved to B[i,j] when (i, j) is in the range of (1:3, 1:4). The notation of ϕ(/i,j/,/4:4,1:4/) is a type of array section descriptor, called segmentation descriptor in our work, to describe the range of indexes and access stride in a data access function. The general form of a segmentation descriptor is defined below.

Definition 1 A **segmentation descriptor**, ϕ specifies a set of accessed array elements, and is defined as follows:

(1)ϕ(/$f_1(i_1,i_2, \cdots ,i_n),\cdots$, $f_m(i_1,i_2, \cdots ,i_n)$/ , /$l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m$/)
={$(i_1,i_2, \cdots ,i_n)$ | $\forall$ k, $1 \le k \le m$, $l_k \le f_k(i_1,i_2,\cdots,i_n) \le u_k$, and there exists a non-negative integer I such that $f_k(i_1,i_2,\cdots,i_n) = l_k + s_k * I$. }

In the form of $l_k{:}u_k{:}s_k$ above, l_k, u_k and s_k indicate the lower bound, upper bound and stride of $f_k(i_1,i_2, \cdots ,i_n)$, respectively. We may omit the stride if it is 1. $f_i(i_1,i_2, \cdots ,i_n)$ is a linear function of $(i_1,i_2, \cdots ,i_n)$.

(2)If α_1 and α_2 are two segmentation descriptors, then the intersection of α_1 and α_2, $\alpha_1 \wedge \alpha_2$, is also a segmentation descriptor. A *simple* segmentation descriptor contains no intersection operators.

$\alpha_1 \wedge \alpha_2 = \{(i_1,i_2, \cdots ,i_n) \mid (i_1,i_2, \cdots ,i_n) \in \alpha_1$ and $(i_1,i_2, \cdots ,i_n) \in \alpha_2 \}$.

The general definition for a data access function is given below.

Definition 2 Let T be an n-dimensional array T and $S_1,\cdots, S_k$ be k arrays. Assume d_i is the number of dimensions in array S_i. A **data access function** has the following form:

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

where

γ_1, γ_2,.., are segmentation descriptors.

$f_1^1, ..., f_1^{d1}, ..., f_k^1, ..., f_k^{dk}, h_1^1, ..., h_k^{dk}$ are linear functions of the index variables, $i_1, ..., i_n$.

$\mathcal{F}1$, $\mathcal{F}2$, ... are elementwise functions.

An expression, $\mathcal{F}1(S_1[f_1^1(i_1,i_2,\cdots,i_n),\cdots,f_1^{d1}(i_1,i_2,\cdots,i_n)],\cdots,S_k[f_k^1(i_1,i_2,\cdots,i_n),\cdots,f_k^{dk}(i_1,i_2,\cdots,i_n)]) \mid \gamma_1$, is called a data access pattern in a data access function. An array operation whose data access function contains multiple data access patterns is a multiple-clause array operation.

Assume $\forall$ i, $1 \le i \le n$, s_i is the size of the i-th dimension of array T. The data access function defined above can be translated to the following pseudo code. If γ_1 is omitted, the data access pattern spans the entire index space of array T.

```
FORALL i1=1 to s1
 FORALL i2=1 to s2
  . . .
  FORALL in=1 to sn
```

```
IF (i1,i2,···,in) ∈ γ1 THEN
```
$\mathrm{T}[i_1,i_2, \cdots ,i_n]= \mathcal{F}1(S_1[f_1^1(i_1,i_2,\cdots,i_n), \cdots, f_1^{d1}(i_1,i_2,\cdots,i_n)], \cdots, S_k[f_k^1(i_1,i_2,\cdots,i_n), \cdots, f_k^{dk}(i_1,i_2,\cdots,i_n)]\ )$

```
IF (i1,i2,···,in) ∈ γ2 THEN
```
$\mathrm{T}[i_1,i_2, \cdots ,i_n]= \mathcal{F}2(S_1[h_1^1(i_1,i_2,\cdots,i_n), \cdots, h_1^{d1}(i_1,i_2,\cdots,i_n)], \cdots, S_k[h_k^1(i_1,i_2,\cdots,i_n), \cdots, h_k^{dk}(i_1,i_2,\cdots,i_n)]\ )$

...

...

Following are a few examples of data access functions:

Example 2 For the EOSHIFT function, the data access function for T=EOSHIFT(A,1,"-1",1) is:

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

Example 3 Let A and B be 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.

Example 4 Function RESHAPE is an example of a data access function between arrays of different dimensional space. The data access function for E=RESHAPE(D,(/2,3/)) is $E[i,j] = D[(j-1)*2+i]$ where $f_1(i,j) = (j-1)*2+i$.

Example 5 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

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

Appendix B gives a summary of data access functions for Fortran 90 array operations which we handle in this paper.

4 Synthesis of Array Operations

Our scheme works by deriving an access function for each array expression or intrinsic array function. We then use a function composition approach to compose those access functions. This composition is semantically equivalent to the synthesis of array functions. Our approach to perform array operation synthesis can be divided into four steps as described below.

Step 1: Build the parse tree of an array expression

The first step is to construct a parse tree for an array expression. Figure 1 shows the parse tree for the running example expression,

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

The source arrays are at the leaf nodes. In this example, arrays A and C are the source arrays. Each internal node including the root corresponds to an array operation. We assign a unique name to each temporary array at an internal node, and the temporary array holds an intermediate result during a straightforward translation. The array labelled as the root is the target array, and it contains the final result of the array expression upon completion.

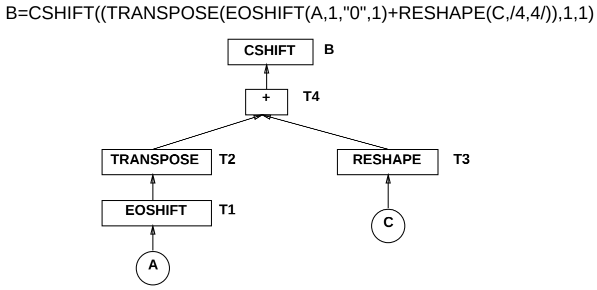

Figure 1: Parse tree of an array expression.

Step 2: Derive the data access functions

In this step, we derive the data access function for each internal node.

For the EOSHIFT function in the running example, the data access function is:

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

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∗4-4], 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

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

Step 3: Synthesize data access functions

In this step, the derived data access functions from step 2 are synthesized into one data access function. A data access function may contain multiple data access patterns delimited by their segmentation descriptors. We first discuss how to synthesize two data access patterns. The synthesis of two data access functions can then be done through a series of syntheses of two data access patterns.

If the segmentation descriptor of a data access pattern is omitted, the access pattern is applied to the whole index space of the target array. Assume the following two multiple-source data access patterns:

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

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

where γ and β are two 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 second equation can be replaced with the composition of $\mathcal{F}2$ and Q_p, where p = 1 $\cdots$ k, with respect to the first equation. It yields the following data access pattern:

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

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

$$\beta' = \phi(/f_1(i_1, i_2, \cdots, i_n), \cdots, f_m(i_1, i_2, \cdots, i_n)/, /l_1 : u_1 : s_1, l_2 : u_2 : s_2, \cdots, l_m : u_m : s_m/). \tag{1}$$

For example, if we have the following two data access patterns: T1[i,j] = A[i,j] | ϕ(/i,j/,/1:3,1:3/) and B[i,j] = $\mathcal{F}$(T1[j,i+1], T3[i+1,j]) | ϕ(/i,j/,/1:3,1:4/). Based on the above synthesis rule, we can derive the synthesized data access pattern as follows:

B[i,j] = $\mathcal{F}$ (A[j,i+1],T3[i+1,j]) | ϕ(/i,j/,/1:3,1:4/) $\wedge$ ϕ(/j,i+1/,/1:3,1:3/)

We now present how to synthesize two data access functions. For the following two data access functions, Equation (3) is the data access function of array S with an x number of data access patterns, and Equation (2) is the data access function of array T with a y number of data access patterns.

$$T[i_1, i_2, \cdots, i_p] = \begin{cases} \mathcal{F}_1(\cdots) & \mid \beta_1 \\ \cdots & \\ \mathcal{F}_q(\cdots) & \mid \beta_q \\ \cdots & \\ \mathcal{F}_y(\cdots) & \mid \beta_y \end{cases} \tag{2}$$

$$S[i_1, i_2, \cdots, i_n] = \begin{cases} \mathcal{G}_1(\cdots, T[\cdots], \cdots) & \mid \gamma_1 \\ \cdots & \\ \mathcal{G}_{k-1}(\cdots, T[\cdots], \cdots) & \mid \gamma_{k-1} \\ \mathcal{G}_k(\cdots, T[\cdots], \cdots) & \mid \gamma_k \\ \mathcal{G}_{k+1}(\cdots, T[\cdots], \cdots) & \mid \gamma_{k+1} \\ \cdots & \\ \mathcal{G}_x(\cdots, T[\cdots], \cdots) & \mid \gamma_x \end{cases} \tag{3}$$

Array T is a temporary array, and T appears in the RHS of the data access patterns of (3). Without loss of generality, we assume that array T appears in all of the data access patterns in (3). In the synthesis of the two data access functions, we have to perform the substitutions of data access patterns $x * y$ times. This will produce a data access function with an $x * y$ number of data access patterns. The final synthesized data access function is in (4):

$$S[i_1, i_2, \ldots, i_n] = \begin{cases} \mathcal{G}_1(\ldots, \mathcal{F}_1(\ldots), \ldots) & \mid \gamma_1 \wedge \beta'_{1,1} \\ \mathcal{G}_1(\ldots, \mathcal{F}_2(\ldots), \ldots) & \mid \gamma_1 \wedge \beta'_{1,2} \\ \vdots & \\ \mathcal{G}_1(\ldots, \mathcal{F}_y(\ldots), \ldots) & \mid \gamma_1 \wedge \beta'_{1,y} \\ \cdots & \\ \mathcal{G}_k(\ldots, \mathcal{F}_1(\ldots), \ldots) & \mid \gamma_k \wedge \beta'_{k,1} \\ \mathcal{G}_k(\ldots, \mathcal{F}_2(\ldots), \ldots) & \mid \gamma_k \wedge \beta'_{k,2} \\ \vdots & \\ \mathcal{G}_k(\ldots, \mathcal{F}_y(\ldots), \ldots) & \mid \gamma_k \wedge \beta'_{k,y} \\ \cdots & \\ \mathcal{G}_x(\ldots, \mathcal{F}_1(\ldots), \ldots) & \mid \gamma_x \wedge \beta'_{x,1} \\ \mathcal{G}_x(\ldots, \mathcal{F}_2(\ldots), \ldots) & \mid \gamma_x \wedge \beta'_{x,2} \\ \vdots & \\ \mathcal{G}_x(\ldots, \mathcal{F}_y(\ldots), \ldots) & \mid \gamma_x \wedge \beta'_{x,y}, \end{cases} \tag{4}$$

where $\beta'_{x,y}$ is calculated in the same way in (1).

The above discussion describes the process of synthesizing two data access functions. To synthesize an array expression or consecutive array operations, we start with the data access function in the root 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 process is repeated top-down on the parse tree until all of the temporary arrays in the RHS are substituted by the source arrays. We use the example in Figure 1 to show how these data access functions are synthesized. We start with the data access function in the root (CSHIFT) of the parse tree:

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

Following is a series of syntheses of two data access functions:

(1) To substitute T4: using T4[i,j]=$\mathcal{F}$(T2[i,j],T3[i,j]) , $\mathcal{F}$(x,y)=x+y

$$\Longrightarrow B[i,j] = \begin{cases} \mathcal{F}(T2[i+1,j], T3[i+1,j]) & | \quad \phi(\text{/i,j/,/1:3,1:4/}) \\ \mathcal{F}(T2[i-3,j], T3[i-3,j]) & | \quad \phi(\text{/i,j/,/4:4,1:4/}) \end{cases}$$

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

(2) To substitute T2: using T2[i,j]=T1[j,i]

$$\Longrightarrow B[i,j] = \begin{cases} \mathcal{F}(T1[j,i+1], T3[i+1,j]) & | \quad \phi(\text{/i,j/,/1:3,1:4/}) \\ \mathcal{F}(T1[j,i-3], T3[i-3,j]) & | \quad \phi(\text{/i,j/,/4:4,1:4/}) \end{cases}$$

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

(3) To substitute T1: using

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

$$\Longrightarrow B[i,j] = \begin{cases} \mathcal{F}(A[j+1,i+1], T3[i+1,j]) & | \quad \phi(\text{/i,j/,/1:3,1:4/}) \wedge \phi(\text{/j,i+1/,/1:3,1:4/}) \\ \mathcal{F}(0, T3[i+1,j]) & | \quad \phi(\text{/i,j/,/1:3,1:4/}) \wedge \phi(\text{/j,i+1/,/4:4,1:4/}) \\ \mathcal{F}(A[j+1,i-3], T3[i-3,j]) & | \quad \phi(\text{/i,j/,/4:4,1:4/}) \wedge \phi(\text{/j,i-3/,/1:3,1:4/}) \\ \mathcal{F}(0, T3[i-3,j]) & | \quad \phi(\text{/i,j/,/4:4,1:4/}) \wedge \phi(\text{/j,i-3/,/4:4,1:4/}) \end{cases}$$

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

(4) To substitute T3: using T3[i,j]=C[i+4*j-4] $\Longrightarrow$

$$B[i,j] = \begin{cases} \mathcal{F}(A[j+1,i+1], C[i+1+4*j-4]) & | \quad \phi(\text{/i,j/,/1:3,1:4/}) \wedge \phi(\text{/j,i+1/,/1:3,1:4/}) \\ \mathcal{F}(0, C[i+1+4*j-4]) & | \quad \phi(\text{/i,j/,/1:3,1:4/}) \wedge \phi(\text{/j,i+1/,/4:4,1:4/}) \\ \mathcal{F}(A[j+1,i-3], C[i-3+4*j-4]) & | \quad \phi(\text{/i,j/,/4:4,1:4/}) \wedge \phi(\text{/j,i-3/,/1:3,1:4/}) \\ \mathcal{F}(0, C[i-3+4*j-4]) & | \quad \phi(\text{/i,j/,/4:4,1:4/}) \wedge \phi(\text{/j,i-3/,/4:4,1:4/}) \end{cases} \tag{5}$$

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

Step 4: Code generation

After having derived the final synthesized data access function (5), we use it to generate a nested loop. For the running example, we use the synthesized data access function obtained in step 3 to generate the pseudo code below.

```
FORALL i=1 to 4
  FORALL j=1 to 4
    If (i,j)∈(φ(/i,j/,/1:3,1:4/) ∧ φ(/j,i+1/,/1:3,1:4/)) THEN B[i,j]=A[j+1,i+1]+C[i+1+4*j-4]
    If (i,j)∈(φ(/i,j/,/1:3,1:4/) ∧ φ(/j,i+1/,/4:4,1:4/)) THEN B[i,j]=C[i+1+4*j-4]
    If (i,j)∈(φ(/i,j/,/4:4,1:4/) ∧ φ(/j,i−3/,/1:3,1:4/)) THEN B[i,j]=A[j+1,i−3]+C[i−3+4*j-4]
    If (i,j)∈(φ(/i,j/,/4:4,1:4/) ∧ φ(/j,i−3/,/4:4,1:4/)) THEN B[i,j]=C[i−3+4*j-4]
```

5 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 may produce significant overhead due to the computation of guarding predicates at run-time. In this section, we develop a systematic scheme to remove the guarding predicates by calculating the exact index ranges of a segmentation descriptor at compile time. The scheme derives the bounds of nested loops to cover the index range of each segmentation descriptor. These bounds are used to generate a loop for each segmentation descriptor without the guarding predicates.

5.1 Optimization for Segmentation Descriptors with Affine Index Functions

In this section, we handle the case that the index function in a segmentation descriptor is an affine function. Assume the following segmentation descriptor,

$\phi(/f_1(i_1,\cdots,i_n),\cdots,f_m(i_1,\cdots,i_n)/,/l_1:u_1:s_1,\cdots,l_m:u_m:s_m/)\wedge\phi^{'}(/f_1^{'}(i_1,\cdots,i_n),\cdots,f_m^{'}(i_1,\cdots,i_n)/,/l_1^{'}:u_1^{'}:s_1^{'},\cdots,l_m^{'}:u_m^{'}:s_m^{'}/)$.

The index functions, f_j, and $f_j^{'}(1\leq j\leq n)$ are affine functions with the form of $C_0 * i{+}C_1$, where C_0 and C_1 are constants, and i is an index variable. There are two steps in this optimization. The first step performs a normalization to transform

$$\phi(/C_0 * i + C_1/,/l:u:m/)$$

into

$$\phi(/i/,/l':u':m'/)$$

where l', u', and m' are the normalized lower bound, upper bound, and stride, respectively.

We then perform the intersection of the segmentation descriptors after the normalization. Below is an example to illustrate the normalization process. The work done on code generations for array assignments [12, 24, 10, 23] in HPF also provide detailed algorithms for this normalization process.

Example 6 Assume a segmentation descriptor, $\phi(/3*i+5/,/4:100:5/)$. We show how to perform normalization.

Step 1. $3*i+5 \in$ (4:100:5) $\Rightarrow 4\leq 3*i+5\leq 100$

Step 2. $3*i+5{=}5*j+4 \Rightarrow 3*i-5*j=-1$. Solve the Diophantine equation $\Rightarrow i=-2-5*t \Rightarrow$ $4{\leq}3*(-2-5*t)+5{\leq}100 \Rightarrow$ -6≤t≤-1

Step 3. i$\in ((-2-5*(-1)):(-2-5*(-6)):|-5|) \Rightarrow$ i$\in$(3:28:5).

The example above shows the steps to normalize the segmentation descriptor for each dimension. By repeating it in all of the dimensions of a segmentation descriptor, $\phi(/f_1(i_1,i_2, \cdots ,i_n),\cdots , f_m(i_1,i_2, \cdots ,i_n)/ , /l_1{:}u_1{:}s_1,l_2{:}u_2{:}s_2,\cdots,l_m{:}u_m{:}s_m/)$, we should get $i_k \in (l'_k{:}u'_k{:}s'_k)$, k=1 to m. Therefore, the normalized segmentation descriptor can be represented as $\phi(/i_1,i_2, \cdots ,i_n)/ , /l'_1{:}u'_1{:}s'_1, l'_2{:}u'_2{:}s'_2, \cdots, l'_n{:}u'_n{:}s'_n/)$.

After having normalized the segmentation descriptors for all dimensions, we then perform intersections of those normalized segmentation descriptors. The process starts with performing an intersection of two segmentation descriptors. The resultant segmentation descriptor is then used to perform an intersection with the next segmentation descriptor.

The intersection can be accomplished by performing an intersection on each dimension independently. That is, ($l_1{:}u_1{:}s_1$, $l_2{:}u_2{:}s_2$, $\cdots$, $l_n{:}u_n{:}s_n$) $\cap$ ($l'_1{:}u'_1{:}s'_1$, $l'_2{:}u'_2{:}s'_2$, $\cdots$, $l'_n{:}u'_n{:}s'_n$) =($l''_1{:}u''_1{:}s''_1$, $l''_2{:}u''_2{:}s''_2$, $\cdots$, $l''_n{:}u''_n{:}s''_n$), $\forall$ i, $1 \leq$ i $\leq$ n, l''_i, u''_i, s''_i are derived from $(l_i{:}u_i{:}s_i) \cap (l'_i{:}u'_i{:}s'_i)$.

An example below describes the steps to perform an intersection, and a detailed algorithm can be also seen in work done on code generations for array assignments[12, 24, 10, 23] in HPF.

Example 7 Assume the following regular section specifiers in two normalized segmentation descriptors: (5:100:6) and (7:200:5). Four steps are needed to perform the intersection of the segmentation descriptors.

Step 1. i $\in$ (5:100:6) $\cap$ (7:200:5) $\Rightarrow$ 7$\leq$i$\leq$100.

Step 2. $i = 6*x+5 = 5*y+7 \Rightarrow 6*x-5*y = 2$. Solve the Diophantine equation $\Rightarrow x = 2-5*t \Rightarrow i = 6*(2-5*t)+5 = -30*t+17$

Step 3. $7 \leq -30*t+17 \leq 100 \Rightarrow -2 \leq t \leq 0$

Step 4. $-30*0+17 = 17$, $-30*(-2)+17 = 77$, *(5:100:6)* $\cap$ *(7:200:5) = (Min(17,77) : Max(17,77) : LCM(6,5)) = (17:77:30)*

Let us consider the segmentation descriptor of the first data access pattern in our example obtained in Step 4 of Section 4, i.e., $\phi(/i,j/,/1:3,1:4/) \wedge \phi(/j,i+1/,/1:3,1:4/))$. After normalization, we obtain $\phi(/i,j/,/1:3,1:4/) \wedge \phi(/i,j/,/0:3,1:3/))$. After intersection, we have $\phi(/i,j/,/1:3,1:3/)$.

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

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

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

These new index ranges can then be used to generate the code below. Figure 2 gives a pictorial view of the composite access patterns after the optimization process. Each parallel loop executes the exact index ranges of the required computation.

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

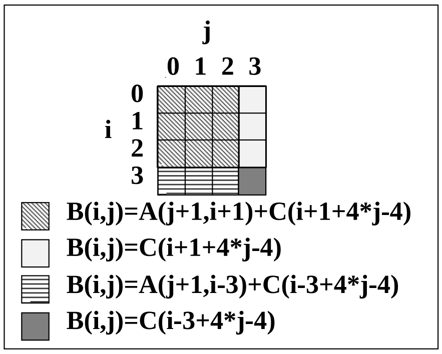

Figure 2: Pictorial view of composite data access patterns after synthesis

```
FORALL j=4 to 4
   B[i,j]=C[i-3+4*j-4]
```

5.2 Optimization for Segmentation Descriptors with Coupled Index Functions

All of the data access functions of the Fortran 90 array operations which we can handle (as listed in Appendix B) can be represented as affine functions except for RESHAPE operations. A non-affine index function can arise when a multiple-clause array operation follows a RESHAPE intrinsic array function. The access function for a RESHAPE operation is a form of coupled index functions. We demonstrate below how to handle such index functions. For example, consider the following code.

```
REAL A(10,100), B(1000), D(1000), E(1000)
B(1:500:1)=D(1:1000:2)
B(501:1000:1)=E(2:1000:2)
A=RESHAPE(B, /10,100/ )
```

We can merge the data access functions of the two array section movements to obtain the following data access function:

$$B[i] = \begin{cases} D[2*i-1] & \mid \phi(/\mathrm{i}/,/1{:}500/) \\ E[2*i-1000] & \mid \phi(/\mathrm{i}/,/501{:}1000/) \end{cases}$$

The data access function of RESHAPE is $A[i,j] = B[i+(j-1)*10] \mid \phi(/\mathrm{i,j}/,/1{:}10,1{:}100/)$. The synthesis produces the following data access function:

$$A[i,j] = \begin{cases} D[2*(i+(j-1)*10)-1] & \mid \phi(/i+(j-1)*10/,/1{:}500/)\wedge\phi(/\mathrm{i,j}/,/1{:}10,1{:}100/) \\ E[2*(i+(j-1)*10)-1000] & \mid \phi(/i+(j-1)*10/,/501{:}1000/)\wedge\phi(/\mathrm{i,j}/,/1{:}10,1{:}100/) \end{cases}$$

To generate code for the first data access pattern, we have to perform a normalization and an intersection on $\phi(/i+(j-1)*10/,/1:500/) \wedge \phi(/i,j/,/1:10,1:100/)$.

The general form of a segmentation descriptor with a coupled index function is $\phi(/a*i+b*j+c/,/l_1{:}u_1{:}s_1/) \wedge \phi(/i/,/l_2{:}u_2{:}s_2/) \wedge \phi(/\mathrm{j}/,/l_3{:}u_3{:}s_3/)$. We show how to generate code based on this form in *Algorithm 1*. An example is listed below to illustrate the steps in *Algorithm 1*.

Example 8 We use this examplis to illustrate *Algorithm 1*. Assume the following segmentation descriptor with one coupled index function: $\phi(/\mathrm{i+(j\text{-}1)*10}/,/1{:}500/) \wedge \phi(/\mathrm{i}/,/1{:}10/) \wedge \phi(/\mathrm{j}/,/1{:}100/)$

`Algorithm 1` : Code generation for segmentation descriptor with a coupled index function
Input: $\phi(/a*i+b*j+c/,/l_1{:}u_1{:}s_1/) \wedge \phi(/i/,/l_2{:}u_2/) \wedge \phi(/j/,/l_3{:}u_3/)$.
Step 1.

For an integer k, we have $a*i+b*j+c = l_1 + s_1*k \Rightarrow a*i+b*j-s_1*k = l_1 - c$.

Use a solver for three variables linear Diophantine equation[1] to solve the above equation (The equation can be solved by 2 GCD calculations). If there is no solution for the above equation, the segmentation descriptor represents an empty set. Assume we get the Diophantine equation solution as follows:

$$\begin{cases} i = p_1*V + p_2*W + c_1 \\ j = q_1*V + q_2*W + c_2 \\ k = r_1*W + c_3 \end{cases} \tag{6}$$

Note that $p_1, p_2, q_1, q_2, r_1, c_1, c_2, and\ c_3$ are integral constants. V and W are integral variables.

Step 2.

Combining Equation (6) with the lower bounds and upper bounds of variables i,j and k, we obtain:

$$l_2 \le p_1*V + p_2*W + c_1 \le u_2 \tag{7}$$

$$l_3 \le q_1*V + q_2*W + c_2 \le u_3 \tag{8}$$

$$l_1 \le l_1 + (r_1*W + c_3)*s_1 \le u_1 \tag{9}$$

By solving inequality (9), it gives the lower bound and upper bound of variable W. Assume L_w and U_w are the lower bound and upper bound of variable W in inequality (9).

Let L_v(x) be the lower bound of V in inequality (7) when W equals x.
Let U_v(x) be the upper bound of V in inequality (7) when W equals x.
Let L'_v(x) be the lower bound of V in inequality (8) when W equals x.
Let U'_v(x) be the upper bound of V in inequality (8) when W equals x.

Step 3.

We then generate parallel loop as follows:

```
FORALL(W= L_w to U_w)
     FORALL (V = MAX(L_v(W), L'_v(W)) to MIN(U_v(W), U'_v(W)) )
```

Note that all of the index variables, i, j and k, within the loop body should be transformed according to (6).

End Of Algorithm

Step 1. For ϕ(/i+(j-1)*10/,/1:500/) $\wedge$ ϕ(/i/,/1:10/) $\wedge$ ϕ(/j/,/1:100/), $i+10*j-10 = 1+k \Rightarrow i+10*j-k = 11 \Rightarrow i = -99+9*W-10*V, j = 11-W+V, k = -W$ by solving three variables linear Diophantine equation[1].

Step 2. $1 \le 1+(-W) \le 500 \Rightarrow -499 \le W \le 0 \Rightarrow L_w = -499\ \ and\ \ U_w = 0$.
$1 \le -99+9*W-10*V \le 10 \Rightarrow \frac{-109+9*W}{10} \le V \le \frac{-100+9*W}{10} \Rightarrow L_v(x) = \left\lceil \frac{-109+9*x}{10} \right\rceil\ and\ U_v(x) = \left\lfloor \frac{-100+9*x}{10} \right\rfloor$.
$1 \le 11-W+V \le 100 \Rightarrow -10+W \le V \le 89+W \Rightarrow L'_v(x) = -10+x\ and\ U'_v(x) = 89+x$.

Step 3. i=-99+9*W-10*V, j=11-W+V, D[2*(i+(j-1)*10)-1]= D[2 * ((-99+9*W-10*V) + ((11-W+V)-1)*10) - 1] = D[1-2*W]. Subsequently, we can generate code as follows:

```
FORALL W= -499 to 0
     FORALL V = MAX(⌈(-109+9*W)/10⌉ ,-10 + W) to MIN(⌊(-100+9*W)/10⌋,89 + W)
          A[-99+9*W-10*V,11-W+V]=D[1-2*W]
```

A segmentation descriptor containing more than one coupled index functions can be solved similarly. We show the form with two coupled index functions below.

$$\phi(/a_1*i+b_1*j+c_1/,/l_1:u_1:s_1/) \wedge \phi(/a_2*i+b_2*j+c_2/,/l_2:u_2:s_2/) \wedge \phi(/i/,/l_3:u_3/) \wedge \phi(/j/,/l_4:u_4/)$$

Algorithm 2 : Code generation for segmentation descriptor with two coupled index functions
Input: $\phi(/a_1$*i+b_1*j+$c_1/,/l_1{:}u_1{:}s_1/) \wedge \phi(/a_2$*i+$b_2$*j+$c_2/,/l_2{:}u_2{:}s_2/) \wedge \phi(/i/,/l_3{:}u_3/) \wedge \phi(/j/,/l_4{:}u_4/)$.
Step 1.

For integers k and l, we have

$$a_1 * i + b_1 * j + c_1 = l_1 + s_1 * k \ \ and \ \ a_2 * i + b_2 * j + c_2 = l_2 + s_2 * l$$

$$\Rightarrow a_1 * i + b_1 * j - s_1 * k = l_1 - c_1 \ \ and \ \ a_2 * i + b_2 * j - s_2 * l = l_2 - c_2$$

Multiplying the first and the second equations by a_2 and a_1, respectively, yields:

$$a_1 * a_2 * i + a_2 * b_1 * j - a_2 * s_1 * k = a_2 * l_1 - a_2 * c_1 \tag{10}$$

$$a_1 * a_2 * i + a_1 * b_2 * j - a_1 * s_2 * l = a_1 * l_2 - a_1 * c_2 \tag{11}$$

Subtract the second equation from the first one, and the result is

$$(a_2 * b_1 - a_1 * b_2) * j - a_2 * s_1 * k + a_1 * s_2 * l = a_2 * l_1 - a_2 * c_1 - a_1 * l_2 + a_1 * c_2$$

Use a solver for three variables linear Diophantine equation[1] to solve the above equation.

Assume that we get the solution of the Diophantine equation as follows:

$$\begin{cases} j = p_1 * V + p_2 * W + c_1 \\ k = q_1 * V + q_2 * W + c_2 \\ l = r_1 * W + c_3 \end{cases} \tag{12}$$

Note that $p_1, p_2, q_1, q_2, r_1, c_1, c_2, and\ c_3$ are integral constants. V and W are integral variables.
Since $a_1 * i + b_1 * j + c_1 = l_1 + s_1 * k$, we have

$$i = \frac{l_1 + s_1 * k - b_1 * j - c_1}{a_1} = \frac{l_1 + s_1 * (q_1 * V + q_2 * W + c_2) - b_1 * (p_1 * V + p_2 * W + c_1) - c_1}{a_1} \tag{13}$$

Step 2.

Combining Equation (12) and (13) with the lower bounds and upper bounds of variables i,j,k and l, we can use the same mechanism in the previous algorithm to compute the range of variable W. Similarly, we can generate a parallel loop.

End Of Algorithm

Algorithm 2 shows the code generation process based on this form. The following example illustrates the steps in the *Algorithm 2*.

Example 9 We use this example to illustrate *Algorithm 2*. Assume the following segmentation descriptor with two coupled index functions: ϕ(/i+(j-1)*10/,/1:500/) $\wedge$ ϕ(/2*i+6*j-51/,/1:600/) $\wedge$ ϕ(/i/,/1:10/) $\wedge$ ϕ(/j/,/1:100/)

Step 1. For ϕ(/i+(j-1)*10/,/1:500/) $\wedge$ ϕ(/2*i+6*j-51/,/1:600/) $\wedge$ ϕ(/i/,/1:10/) $\wedge$ ϕ(/j/,/1:100/),
Combining $i + 10 * j - 10 = 1 + k$ and $2 * i + 6 * j - 51 = 1 + l$, we have $14 * j - 2 * k + l = -30$. $\Rightarrow$ j=V, k=15-W+7*V, l=-2*W, i=11-10*j+k=11-10*V+(15-W+7*V)=26-3*V-W

Step 2. $1 \le i \le 10, 1 \le j \le 100, 0 \le k \le 499, 0 \le l \le 599$

$$\Rightarrow \begin{cases} 1 \le 26 - 3 * V - W \le 10 & \Rightarrow \lceil \frac{-16+W}{-3} \rceil \le V \le \lfloor \frac{-25+W}{-3} \rfloor \\ 1 \le V \le 100 & \Rightarrow 1 \le V \le 100 \\ 0 \le 15 - W + 7 * V \le 499 & \Rightarrow \lceil \frac{-15+W}{7} \rceil \le V \le \lfloor \frac{484+W}{7} \rfloor \\ 0 \le -2 * W \le 599 & \Rightarrow -299 \le W \le 0 \end{cases}$$

Based on these index ranges, a parallel loop is generated as follows:

```
FORALL W= -299 to 0
     FORALL V = MAX(⌈(-16+W)/(-3)⌉, 1, ⌈(-15+W)/7⌉) to MIN(⌊(-25+W)/(-3)⌋, 100, ⌊(484+W)/7⌋)
         ...
```

In the above algorithm, we first eliminate index variable i and then solve a linear Diophantine equation with index variables j, k, and l. The bounds of i, j, k, and l are then used to construct the ranges of loop index variables V and W. The above method can be extended to segmentation descriptors with more than two coupled index functions. One can first eliminate index variables by subtracting equations as done in the the above algorithm, and then solve a linear Diophantine equation. The bounds of index variables are then used to construct the ranges of loop index variables V and W. The index function of m variables can be solved by solving an $m+1$ variables Diophantine equation based on the result in classic elementary number theory [1]. However, we have not found a need of a segmentation descriptor with three or more coupled index functions in real applications.

6 Synthesis Anomaly and Solution

Array synthesis can improve program performance by reducing redundant data movement, temporary storage usage, and loop barrier synchronization overhead. The proposed 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 being considered to be eliminated is replicated by a one-to-many array operation or is a program temporary which will be used multiple times. Note that in the absence of common subexpressions and one-to-many array operations, the array operation synthesis always reduces the amount of stores, loads, and computation in consecutive array operations. In the case of a program temporary which will be used multiple times, the synthesized data access functions in the whole program may contain common subexpressions. This can be solved by extending the conventional method mentioned in [2] to eliminate common subexpressions. What remains to be solved is the replication problem in one-to-many operations. The typical one-to-many array operation in Fortran 90 is the SPREAD intrinsic array function. For example, consider the following code segment:

```
REAL A(N),B(N),C(N),T(N,M),D(N,M)
A=SIN(SQRT(B+0.5)+COS(C)) ··· (S1)
T=SPREAD(A,DIM=2,NCOPIES=M)+D ··· (S2)
```

Assume that array A above is a temporary variable to be eliminated by the synthesis process. If we synthesize array expressions **(S1)** and **(S2)** separately, the performance will be better than that of synthesizing array expressions **(S1)** and **(S2)** collectively based on the presented schemes so far. In the first case which synthesizes the expressions separately, the synthesized code is as follows:

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

It needs N times of each of SIN, SQRT, and COS arithmetic operations, $2*N+N*M$ times of floating-point addition operations, and $N+N*M$ times of assignments. In the second case which synthesizes the expressions collectively, **(S1)** and **(S2)** are synthesized collectively, the synthesized code is as follows:

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

It needs $N*M$ times of each of SIN, SQRT, and COS arithmetic operations, $3*N*M$ times of floating-point addition operations and $N*M$ times of assignments. In virtually all computer systems, the run time performance from the first case will outperform that of the second one. This example shows that the synthesis may increase the cost of total computation. This is due to the fact that the SPREAD function in Fortran 90 replicates an array by adding one more dimension. This function specifies a one-to-many data movement between the source and target arrays.

6.1 Prevention of Synthesis Anomaly

One way to avoid this performance anomaly is to detect every appearance of one-to-many operation, and use it as a boundary for the synthesis scheme. However, this conservative method does not exploit the full opportunities for synthesis optimization. In this section, we present a heuristic method which can synthesize an important subset of expressions containing one-to-many operations, without incurring any performance anomaly. Our algorithm is listed in *Algorithm 3*. It can be divided into three steps. In the first step, it checks if any one-to-many operation exists. If the one-to-many function does not exist, array operation synthesis is performed based on the earlier scheme. If there is a one-to-many operation, the expression which substitutes the source of a one-to-many array operation is placed within a pair of curly brackets. In our SPREAD example, we get the following data access pattern:

T[i,j]={SIN(SQRT(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 curly brackets from the final synthesized expression if the expression within a pair of curly brackets does not have any arithmetic computation. The rationale is that in this case there is no computation to be replicated.

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

Definition 3 Assume S = $\{a_1,a_2,\cdots,a_n\}$ is a sequence of an 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 $a_1,a_2,\cdots,a_k$.

For example, {i}, {j, i}, and {i, j, k} are admissible prefixes of {i,j,k}, but {j} and {j, k} are not. Following is an example to illustrate the steps in *Algorithm 3*.

Example 10 Suppose we have a program as follows.

```
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(SQRT(B+0.5)+SPREAD(G,DIM=2,NCOPIES=M))
T=SPREAD(A,DIM=3,NCOPIES=L)+D+SPREAD(E,DIM=3,NCOPIES=L )
```

Step 1. We get the synthesized code as follows: T[i,j,k] = {SIN(SQRT(B[i,j]+0.5) + {COS(C[i])})} + D[i,j,k] + {E[i,j]}

Step 2. Eliminate the pair of curly brackets of E in this case and get T[i,j,k] = {SIN(SQRT(B[i,j]+0.5) + {COS(C[i])})} + D[i,j,k]+E[i,j].

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

```
FORALL i=1 to N
```

Algorithm 3 : Synthesis anomaly prevention

Input: A sequence of array statements to be synthesized.
(Assume that the target array at the LHS of the synthesized data access function is T[i_1,i_2,$\cdots$,i_n])
Step 1.

IF (there exists no one-to-many array operation)

Perform array synthesis and code generation as defined in previous sections.
EXIT.

ELSE

Perform a 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 curly brackets.

ENDIF

Step 2.

Remove the pair of curly brackets from the final synthesized expression if the expression within the curly brackets does not have any arithmetic computation.

Step 3.

IF (there exists an array subscript within the curly brackets in the RHS of the synthesized data access function which is not an admissible prefix of $\{i_1,i_2,\cdots,i_n\}$)

Report the statement number, where the expressions inside and outside the curly 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 $i_1,i_2,\cdots,i_k$.
(2) Create a distinct 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 i_k.
(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.

ENDIF

End Of Algorithm

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

6.2 Loop Interchange for More Synthesis

In the mechanism proposed in the previous section, the synthesis process has to stop when there is a one-to-many operations and the array subscript within a pair of curly brackets created in the Algorithm 3 is not an admissible prefix of the index variables of the target array. In this section, we allow continuation of the synthesis process by devising the loop orders so that the array subscript within a pair of curly brackets can be an *admissible prefix* of the index variables of the target array.

For example, if we have the following data access function.

$T[i,j,k] = \{COS(B[j,k])\} + E[i,j,k] * \{SIN(D[k])\}$

Since {j,k} and {k} are not admissible prefixes of {i,j,k}, thus the synthesis stopped at the one-to-many data access function. However, both {j,k} and {k} are admissible prefixes of {k,j,i}. Therefore, we can rearrange the loop nested order to {k,j,i} instead of {i,j,k}. The loop interchange makes the synthesis possible. In this example, we can find that both {k} and {j,k} are admissible prefixes of {k,j,i}. We rearrange the loop nested order to k,j,i and generate the synthesized code without duplicated computation as follows.

```
FORALL k=1 to L3
   Temp1=SIN(D[k])
   FORALL j=1 to L2
      Temp2=COS(B[j,k])
      FORALL i=1 to L1
         T[i,j,k]=Temp2+E[i,j,k]*Temp1
```

7 Advanced Synthesis Techniques

In this section, we discuss synthesis beyond a single statement and point out the constrains of our synthesis scheme. In addition, we present two advanced synthesis techniques. The first is to synthesize array reduction and location operations. The second is the method to synthesize WHERE constructs of Fortran 90.

7.1 Global Synthesis and Synthesis Constraints

Based on the statement merge optimization in [16], this synthesis framework can be applied to synthesize the array operations in different array assignment statements. For example, if T is a program 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. Statement merge needs accurate data flow analysis of the reaching definition and use information and is well treated in [16]. The main issue is to decide whether an array is a temporary array or not. To obtain this information, we need to perform liveness analysis [2]. For example, see the following code fragment which is in the inner-most loop of Sandia Wave application [16].

```
S1:      FXP = CSHIFT(F1,1,+1)
S2:      FXM = CSHIFT(F1,1,-1)
S3:      FYP = CSHIFT(F1,2,+1)
S4:      FYM = CSHIFT(F1,2,-1)
S5:      FDERIV = ZXP * (FXP - F1) + ZXM * (FXM - F1) +
        ZYP * (FYP - F1) + ZYM * (FYM - F1)
```

The dataflow analysis for reaching definitions finds out that the definitions of FXP, FXM, FYP and FYM in S1, S2, S3 and S4, respectively, reaches S5. The dataflow analysis for liveness finds out that FXP, FXM, FYP and FYM are not used anywhere other than statement S5. Thus, FXP, FXM, FYP and FYM are array temporaries for S5. The synthesis continues until all of the temporary arrays are eliminated.

Our framework can handle an extensive set of array operations include TRANSPOSE, CSHIFT, EOSHIFT, MERGE, SPREAD, RESHAPE, array reduction and location functions, and WHERE constructs. In the case of array section assignment operations, we have two constraints to synthesize multiple array assignment statements, which refer to the same array but different sections as their targets. First, these array section assignments may not move data into overlapped areas. Second, the union of different sections at LHS in multiple statements is equal to the whole array. These two constraints can be checked statically. Synthesis with array elements referenced in DO loops and FORALL loops is beyond the scope of this paper.

7.2 Synthesis of Array Reduction and Location Functions

A reduction function *reduces* an argument array in the way that all elements along a specified dimension participate in producing a scalar result and thus the number of dimensions is reduced by one. A location function locates the maximum or minimum value in the argument array along a specified dimension. Because of the many-to-one property of array reduction and location functions, they are difficult to represent the data access functions presented in the preceding sections. For example,

```
real a(50,100), e(50)
e=sum(a,dim=2)
```

For the above array reduction function SUM, the data access function may be specified as:

$$e[i]=\mathcal{F}(a[i,1],a[i,2],\cdots,a[i,99],a[i,100])$$

$$\mathcal{F}(\alpha_1,\alpha_2,\cdots,\alpha_{99},\alpha_{100})=\alpha_1+\alpha_2+\cdots+\alpha_{99}+\alpha_{100}$$

Although it is possible to represent a data access function in this manner, yet the number of arguments of $\mathcal{F}$ is 100. When an array is large, it is often impractical to represent an array reduction function using the original form of data access function. Thus, we propose an **extended data access function** as follows:

e[i]=$SUM_{j=1:100:1}(a[i,j])$

The regular section specifier specifies that a reduction operation is applied to the second dimension, and the range is from 1 to 100 with stride 1.

The synthesis of extended data access functions is similar to the synthesis of original data access functions. The only difference is to retain the regular section specifier in the synthesis process. We use an example below to illustrate the synthesis of extended data access functions. The following example is extracted from [11].

```
real a(50,100),b(100),c(100,200)
real d(100,50),e(50),f(50),g(50)
real t1(50,100), t2(50,100),t3(100,50),t4(100,50),t5(50),t6(50)
d=c(:,1:200:4)
t1=transpose(d)
t2=a+t1
e=sum(t2,dim2)
t3=spread(b,dim=2,ncopies=50)
t4=t3+d
f=sum(t4,dim=1)
t5=b(1:100:2)
t6=t5+e
g=t6+f
```

There are ten array operations in the above code fragment. Arrays d, e, f, t1, t2, t3, t4, t5, and t6 are temporaries which can be eliminated by array operation synthesis assuming that they are not used elsewhere. We list their corresponding data access functions below. Note that some of array operations are array reduction operations which are represented with our extended data access functions.

d[i,j]=c[i,j*4] $\cdots$ (A)

t1[i,j]=d[j,i] $\cdots$ (B)

t2[i,j]=a[i,j]+t1[i,j] $\cdots$ (C)

e[i]=$SUM_{j=1:100:1}(t2[i,j])$ $\cdots$ (D)

t3[i,j]=b[i] $\cdots$ (E)

t4[i,j]=t3[i,j]+d[i,j] $\cdots$ (F)

f[i]=$SUM_{j=1:100:1}(t4[j,i])$ $\cdots$ (G)

t5[i]=b[2*i] $\cdots$ (H)

t6[i]=t5[i]+e[i] $\cdots$ (I)

g[i]=t6[i]+f[i] $\cdots$ (J)

The following process shows the steps of synthesizing the ten operations:

By synthesizing (H), (I) and (J), we have:

g[i]=b[2*i]+e[i]+f[i]

Using (D) and (G) to substitute arrays f and e, we get:

g[i]=b[2*i]+$SUM_{j=1:100:1}(t2[i,j])+ SUM_{j=1:100:1}(t4[j,i])$

Using (C) and (F) to substitute arrays t2 and t4, we get:

g[i]=b[2*i]+$SUM_{j=1:100:1}(a[i,j]+t1[i,j])+ SUM_{j=1:100:1}(t3[j,i]+d[j,i])$

Using (B) and (E) to substitute arrays t1 and t3, we get:

g[i]=b[2*i]+$SUM_{j=1:100:1}(a[i,j]+d[j,i])+ SUM_{j=1:100:1}(b[j]+d[j,i])$

Using (A) to substitute array d, we get:

g[i]=b[2*i]+$SUM_{j=1:100:1}(a[i,j]+c[j,4*i])+ SUM_{j=1:100:1}(b[j]+c[j,4*i])$

If the targeted system supports a library call for reduction function SUM, we can generate code as follows:

```
FORALL i=1 to 50
  g[i]=b[2*i]+SUM_{j=1:100:1}(a[i,j]+c[j,4*i])+ SUM_{j=1:100:1}(b[j]+c[j,4*i])
```

After eliminating common subexpressions and moving loop-invariant computation out of the loop, we obtain the following code:

```
TEMP=SUM_{j=1:100:1}(b[j])
FORALL i=1 to 50
  g[i]=b[2*i]+SUM_{j=1:100:1}(a[i,j])+ 2*SUM_{j=1:100:1}(c[j,4*i])+ TEMP
```

The example above shows the extension of our synthesis scheme to include reduction functions. If the source array of a reduction operation comes from a data access function with multiple data access patterns, the synthesis is more complicated and has to be treated carefully. Appendix C presents a solution for the general case.

7.3 Synthesis of WHERE Constructs

Based on the statement merge optimization in [16], this synthesis framework can be applied to synthesize the array operations in different array assignment statements. 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 a synthesis between the derived data access function and the data access functions of enclosed array statements. In addition, 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. The data access functions of both bodies of WHERE and ELSE-WHERE constructs need to have the same target array to be considered in our scheme. The data access function of the combined WHERE and ELSE-WHERE statement can then be synthesized with array statements outside the WHERE statement. An example is given 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

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

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

8 Complexity

We analyze the complexity of our synthesis scheme in this section. The time for synthesizing two data access patterns is bounded by O($x*y*D$), where D is the largest number of the dimensions of source arrays, and x and y are the numbers of data access patterns of the two data access functions, respectively. Assume there are k data access functions, $F_1, F_2, \cdots, F_k$, and l_i is the number of data access patterns of F_i. The time of synthesizing, $F_1, F_2, \cdots, F_k$, is

$l_1*l_2*D+(l_1*l_2)*l_3*D+(l_1*l_2*l_3)*l_4*D+\cdots+(l_1*l_2*l_3*l_4*\cdots*l_{k-1})*l_k*D \leq (l_1*l_2*l_3*l_4*\cdots*l_{k-1})*l_k*D*(k-1) = O(D*k*\prod_{i=1}^{k} l_i)$.

The nature of our array synthesis is a form of local optimization in contrast to the global optimization in a compiler. In addition, by examining many real Fortran 90 programs, we have found that the number of consecutive array operations within an array expression is uaually less than 12. The data access functions of Fortran 90 intrinsic array functions usually have the number of data access patterns less than four. In fact, many array operations, including TRANSPOSE, array section move, SPREAD, etc., have only one data access pattern. The complexity of synthesis and optimization of array operations is exponential and may sound impractical if k is large. However, in real application, we can set the maximum synthesis length for multiple-clause functions to be a fixed constant, say 10 or 12. This makes our algorithm tractable for a compiler to use.

The time of performing optimization for segmentation descriptor is the combination of the normalization and intersection processes as mentioned in Section 5. For the normalization, we have to normalize all of the segmentation descriptors in the data access patterns of the resultant data access function. Because the resultant data access function is from the synthesis of k data access functions, we also need to solve the intersection of at most k simple segmentation descriptors. The running time for normalizing a simple segmentation descriptor is bounded by the running time of solving D linear Diophantine equations, where D is the number of dimensions of the array. Similarly, the running time for solving the intersection of two normalized segmentation descriptors with affine index functions is also bounded by solving D linear Diophantine equations. Therefore, The overall running time for normalizing and intersecting the segmentation descriptors of $F_1, F_2, \cdots, F_k$, (and l_i is the number of data access patterns of F_i), is

$$O(l_1 * l_2 * l_3 * l_4 * \cdots * l_k * D * \sigma * k),$$

where σ is the time to solve a Diophantine equation, and σ is equal to $(Log\ c)$, where c is an integer produced in finding the greatest common divisor during the equation solving process.

9 Implementation and Experiments

The SYNTOOL is a preliminary implementation of this synthesis scheme, and it provides a prototype for the experiments in this section. It is written in C language and can be installed on any system with a native C compiler. Users can use it to construct their own data access functions. The SYNTOOL provides a routine to synthesize those user-specified data access functions. Also, the routines for normalization and intersection of segmentation descriptors are also included in the SYNTOOL. The SYNTOOL takes data access functions as inputs and performs an automatic array operation synthesis to obtain a synthesized data access function as discussed in this paper.

Seven Fortran 90 programs with array operations shown in the appendix A are used to evaluate the effectiveness of our synthesis scheme in improving execution time performance. 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 applied in the base version so that there is only one single loop generated without any temporary array for a sequence of elemental array operations, such as A(:) = B(:) + C(:) + D(:). The synthesized codes are obtained by feeding data access functions of array operations into SYNTOOL to automatically construct synthesized data access functions, which are then used to generate C code. This code generation process is done by hand. The execution time shown in Table 1 is obtained by running each program 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, and CSHIFT functions as well as WHERE and ELSE-WHERE constructs. They are used to demonstrate the ability of our scheme in composing these array operations and to show the performance effects. The first test case is the example used in Section 4. The second example includes WHERE and ELSE-WHERE constructs and a SPREAD function. The third example

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 multiprocessor machine with 8 nodes. The performance improvement is measured with the execution time of the base version over the synthesis version on an equal number of processors.

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 extracted from real applications. The dimension size of the data grid is 256. The execution times of suites 4, 5 and 6 are measured by running the program 20 times. Test suite 4 is the inner most loop in the Problem 9 of Purdue Set [22]. 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[21]. The performance is improved by more than 20 percent regardless of the number of processors. In addition, we employ the solution to address the performance anomaly problem in this test case. The code fragment in the sixth suite is modified from the Sandia Wave application [16]. The Sandia Wave program solves a 2-D wave proram in a periodic domain with an obstacle. It uses a leap-grag scheme and second order spatial diferences. We extracted the codes from the version originally developed for CM-2 machine. The performance is improved by nearly three times on eight processors. The last code segment is from the Linear Equation Solver in the book by Brainerd et al.[5]. This program shows improvements up to 2.7 folds.

We then compare our synthesis scheme with the scheme of Guibas and Wyatt [14] and Budd [6], which synthesize 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, their work 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 certain special cases of RESHAPE. However, for the constructs, such as CSHIFT, EOSHIFT, MERGE, and WHERE, which cannot be expressed by 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

Time (seconds)	p=1	p=2	p=4	p=8
(4)[a] (PURDUE-Set Prob9) base code	84.90	44.39	24.09	17.35
(4)[b] Synthesized code without loop guard optimization	19.76	10.24	5.88	3.79
(4)[d] Final synthesized code	14.09	5.88	4.29	3.19
(5)[a] (Magnetic E.) base code	11.02	5.73	3.23	2.01
(5)[c] synthesis code without eliminating anomaly	10.55	5.29	2.92	1.67
(5)[d] Final synthesized code	7.33	3.68	2.01	0.99
(6)[a] (Sandia Wave)	40.95	25.14	13.36	8.55
(6)[b] Synthesized code without loop guard optimization	15.78	9.55	5.07	3.12
(6)[d] Final synthesized code	11.21	6.85	3.68	2.38
(7)[a] (Templates from Solver)	108.14	55.55	33.86	26.89
(7)[d] Final synthesized code	40.72	22.64	13.93	11.68

Table 2: Performance of test suites on an SGI Power Challenge with 8 nodes. The problem size N is set to 512.

1, 2, 3, 4, 6, and produces the same codes as our base version. The stepper mechanism only successfully synthesize the program of the test suite 7, and produces the same synthesized code 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. Note that performance anomaly does occur with the test suite 5. The solution in Section 6 enables us to produce better optimized codes by avoiding performance anomaly problems.

We also perform experiments using the same programs on an 8-node SGI Power Challenge machine, and the results are shown in Table 2. The grid size is set to 512 on this platform, and the time is measured in seconds for the code fragments executed 20 times. We only list information for the four real applications, as they are of more interests. We also measure the effects to perform synthesis with or without the segmentation descriptor optimization (described in Section 5) and the synthesis anomaly prevention scheme (described in Section 6). Four sets of performance data are given for each application, and they are named item [a], [b], [c], and [d] in Table 2. The first set (item [a]) of the performance figures given for each application is from the base code. The second set (item [b]) of the performance data includes array synthesis but without segmentation descriptor optimization and synthesis anomaly handling. The third set (item [c]) of the performance data includes array synthesis and segmentation descriptor optimization, but without synthesis anomaly handling. The last set (item [d]) of the performance data includes array synthesis with descriptor optimization and synthesis anomaly prevention. The second and third items described above might be omitted if there is no need for such an optimization for an application. In the case that the synthesized result is with only one pattern in the synthesized data access function, there is no need for array descriptor optimization. In addition, in the absence of common subexpressions and one-to-many array operations, there is no need for synthesis anomaly handling. Our scheme is equally effective on an SGI Power Challenge machine. Table 3 illustrates the benefit of our optimization. The performance is normalized so that the base code running on one processor for each application is set to one. Each given speedup is the time of the corresponding entry in Table 2 over the base code running on one processor.

Finally, Table 4 lists the number of stores, loads, barrier synchronizations, and array storage used in the base and synthesized version, respectively. N is the size of the array in each dimension. The last column in the table lists the number of total words for arrays used in the test suites. In this table, we only count the numbers of loads

Time (seconds)	p=1	p=2	p=4	p=8
(4)[a] (PURDUE-Set Prob9) base code	1	1.9	3.5	4.9
(4)[b] Synthesized code without loop guard optimization	4.3	8.2	14.4	22.4
(4)[d] Final synthesized code	6.0	14.4	19.7	26.6
(5)[a] (Magnetic E.) base code	1	1.92	3.41	5.48
(5)[c] synthesis code without eliminating anomaly	1.04	2.08	3.77	6.59
(5)[d] Final synthesized code	1.50	2.99	5.48	11.13
(6)[a] (Sandia Wave)	1	1.62	3.06	4.78
(6)[b] Synthesized code without loop guard optimization	2.59	4.28	8.07	13.12
(6)[d] Final synthesized code	3.65	5.97	11.12	17.20
(7)[a] (Templates from Solver)	1	1.94	3.19	4.02
(7)[d] Final synthesized code	2.65	4.77	7.76	9.25

Table 3: Performance Ratio on an SGI Power Challenge with 8 nodes. The problem size N is set to 512.

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 4: Number of loads, stores, barrier synchronizations, and array storages used in test suites

and stores for array elements, as they are the main factors contributing to improvements in the synthesis schemes. The numbers of data movements are significantly reduced in all cases, which is the main reason for performance improvement. In addition, the number of loop barriers and array storage used are all reduced, which, too, contribute to performance improvement.

10 Conclusion

Array operation synthesis transforms multiple array operations into a single data access function. Thus, the synthesis can improve performance by reducing redundant data movements, temporary storage usage, and parallel loop synchronization overhead. In this paper, we have presented a new function composition approach to synthesize extensive array operations. Our scheme can handle not only data accesses between arrays of different numbers of dimensions but also masked array expressions, multiple-source array operations, and multiple-clause array functions, which are not handled collectively by any previous research work. As a result, we can handle compositions of RESHAPE, SPREAD, EOSHIFT, TRANSPOSE, CSHIFT, and MERGE functions, array section move, array reduction operations, WHERE constructs, and ELSE-WHERE constructs in Fortran 90. 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] Harriet Griffin. *Elementary Theory of Numbers*, McGraw-Hill Book Company, Inc, 1954.

[2] Afred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[3] 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.

[4] Vasanth Balasundaram and Ken Kennedy. A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations, Proceedings of ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, pp. 41-53, June 1989.

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

[6] 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.

[7] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. Size and Access Inference for Data-Parallel Programs, Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, pp. 130-144, June 1991.

[8] S. Chatterjee, J. R. Gilbert, and R. Schreiber. *Mobile and Replicated Alignment of Arrays in Data-Parallel Programs*, Proceedings of Supercomputing '93, Portland, pages 420-429, Nov. 1993,

[9] Siddhartha Chatterjee. *Compiling Nested Data-Parallel Programs for Shared-Memory Multiprocessors*, ACM Transactions on Programming Languages and Systems, 15(3), pages 400-462, July 1993.

[10] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating Local Addresses and Communication Sets for Data Parallel Programs. *Proceedings of Fourth ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, pages 149–158, May 1993.

[11] Siddhartha Chatterjee, John R. Gilbert, R. Schreiber, and Shang-Hua Teng, Automatic Array Alignment in Data-Parallel Programs, Proceedings of the Twentieth Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, SC, pages 16-28, January 1993.

[12] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C. H. Huang, and P. Sadayappan. On Compiling Array Expressions for Efficient Execution on Distributed-memory Machines. *In Proceedings of ICPP*, pages 301–305, 1993.

[13] Gwan-Hwan Hwang, Jenq Kuen Lee, and Dz-ching Ju. An Array Operation Synthesis Scheme to Optimize Fortran 90 Programs, Proceedings of ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, pp. 112-122, July 1995.

[14] 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.

[15] 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.

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

[17] Paul Havlak and Ken Kennedy. *An Implementation of Interprocedural Bounded Regular Section Analysis*, IEEE Trans. Parallel and Distributed Systems, 2(3):350-360, July 1991.

[18] Kathleen Knobe and William J. Dally. *The Subspace Model: A Theory of Shapes for Parallel Systems*, Workshop on Automatic Data Layout and Performance Prediction, Rice University, Houston Texas, April 1995.

[19] C. Koelbel , D. Loveman, R. Schreiber, G. Steele and M. Zosel. *The High Performance Fortran Handbook*, MIT-press, Cambridge, 1994.

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

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

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

[23] Edwin M. Paalvast, Henk J. Sips, and A. J. van Gemund. *Automatic Parallel Program Generation and Optimization from Data Decompositions*, Proceeding of ICPP, 1991.

[24] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating Communication for Array Statements: Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing*, pages 150–159, 1994.

[25] Richard C. Waters *Automatic Transformation of Series Expressions into Loops*, ACM Transactions on Programming Languages and Systems, Vol.13, No.1, Pages 52-98, Jan. 1991.

[26] Clifford Walinsky and Deb Banerjee, *A Data Parallel FP Compiler*, Journal of Parallel and Distributed Computing 22, 138-153, 1994.

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

A Code Fragments for Experiments

1.
```
T=A
B=CSHIFT(TRANSPOSE(T+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.

Section movement, Fortran 90 standard intrinsic function

A(l_1:u_1:s_1)=B(l_2:u_2:s_2)

The data access function is A(l_1+s_1*i)=B(l_2+s_2*i) | ϕ(/i/,/0:$\frac{u_1-l_1}{s_1}$:1/)

TRANSPOSE, 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, Fortran 90 standard intrinsic function : circular shift of the elements of the argument array

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

$If\ \ l > 0\ \Rightarrow$

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

$If\ \ l < 0\ \Rightarrow$

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

"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.

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

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

EOSHIFT, 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)

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

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

MERGE, 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, 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, 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,

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

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

(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:

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

Array Reduction Functions, Fortran 90 standard intrinsic function:

See section 7.2.

WHERE constructs: Masked Array Assignment statements of Fortran 90

See section 7.3.

The following are not Fortran 90 standard intrinsic functions.

REVERSE, 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.

CONCATENATION, concatenates two array along a specified axis (There exists such an array operator in APL.)

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

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

"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, array section move, array reduction operations 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 have not been included in this framework.

C Supplements to Synthesis with Reduction Operation

In the case that the source array of a reduction operation comes from a multiple-clause data access function, the synthesis with reduction becomes more complicated and has to be carefully handled. We explain the idea of array operation synthesis involved in multiple-clause data access functions and array reduction operations below. Consider the following Fortran 90 code fragment:

```
T1(1:50,1:40)=A(1:50,1:40)
T1(51:100,1:60)=B(1:50,1:60)
T1(1:50,41:50)=C(1:50,1:10)
T1(1:20,51:100)=D(1:20,1:50)
T1(21:50,51:60)=E(1:30,1:10)
T1(21:60,61:100)=F(1:40,1:40)
T1(61:100,61:100)=G(1:40,1:40)
E=SUM(H+TRANSPOSE(T1),dim=2)
```

Since arrays A, B, C, D, E, F and G are assigned to different portions of array T1, their data access functions can be merged into a single data access function. We can construct the data access functions below.

$$T1[i,j] = \begin{cases} A[i,j] & \mid \phi(/\text{i,j}/,/1{:}50,1{:}40/) \\ B[i-50,j] & \mid \phi(/\text{i,j}/,/51{:}100,1{:}60/) \\ C[i,j-40] & \mid \phi(/\text{i,j}/,/1{:}50,41{:}50/) \\ D[i,j-50] & \mid \phi(/\text{i,j}/,/1{:}20,51{:}100/) \\ E[i-20,j-50] & \mid \phi(/\text{i,j}/,/21{:}50,51{:}60/) \\ F[i-20,j-60] & \mid \phi(/\text{i,j}/,/21{:}60,61{:}100/) \\ G[i-60,j-60] & \mid \phi(/\text{i,j}/,/61{:}100,61{:}100/) \end{cases} \tag{14}$$

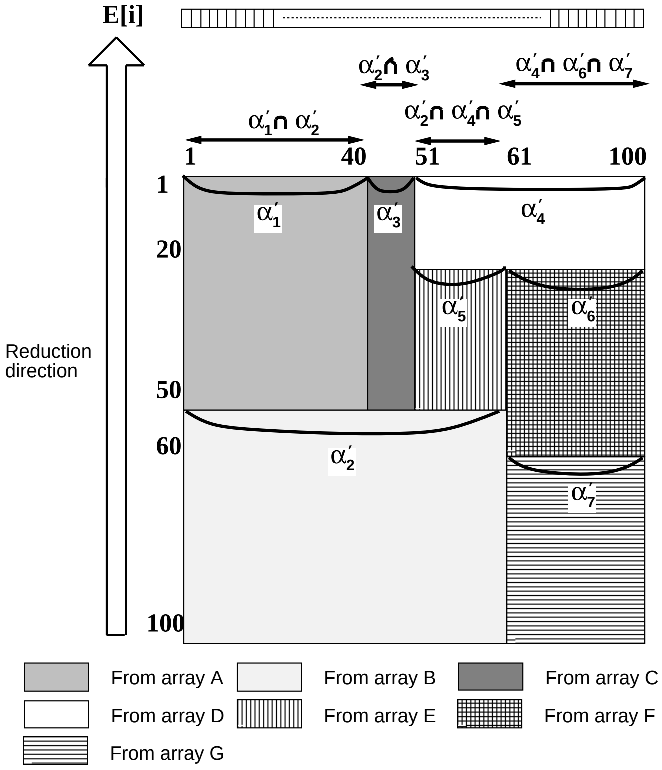

Figure 3: Data layout of array T1

$$T2[i,j] = T1[j,i] \tag{15}$$

$$T3[i,j] = H[i,j] + T2[i,j] \tag{16}$$

$$E[i] = SUM_{j=1:100:1}(T3[i,j]) \tag{17}$$

After synthesizing (15), (16), and (17), we have

$$E[i] = SUM_{j=1:100:1}(H[i,j] + T1[j,i]). \tag{18}$$

Note that the data access function of array T1 is a multiple-clause one. See (18), E[i] comes from the reduction of the ith *row* of array H and the ith *column* of array T1. This is because array T1 is first transposed before the reduction operation. *Fig. 3* shows that the sources for different columns in array T1 vary according to the value of i. In the case that i is in the range from 1 to 40, the ith column is from array A and B. Similarly, when i is in the range from 41 to 50, the ith column is from array C and B, and if i is in the range from 51 to 60, the ith column is from array D, E, and B. The source arrays are D, F and G if i is in the range from 61 to 100. The main idea is to group the sections with the same reduction patterns together. We can get those information from the segmentation descriptors of the synthesized data access function. With those segmentation descriptors, we perform intersections to obtain the index ranges which have the same reduction patterns.

The whole process can be divided into several steps. We use the running example to illustrate these steps.

Step 1. Synthesize data access functions as described in Section 7.2 In our example, we synthesize (14), (15), (16), and (17) together.

$$
E[i] = \begin{cases}
SUM_{j=1:100:1}(H[i,j] + A[j,i]) & \mid \phi(\texttt{/i,j/,/1:40,1:50/}) \rightarrow \alpha_1 \\
SUM_{j=1:100:1}(H[i,j] + B[j-50,i]) & \mid \phi(\texttt{/i,j/,/1:60,51:100/}) \rightarrow \alpha_2 \\
SUM_{j=1:100:1}(H[i,j] + C[j,i-40]) & \mid \phi(\texttt{/i,j/,/41:50,1:50/}) \rightarrow \alpha_3 \\
SUM_{j=1:100:1}(H[i,j] + D[j,i-50]) & \mid \phi(\texttt{/i,j/,/51:100,1:20/}) \rightarrow \alpha_4 \\
SUM_{j=1:100:1}(H[i,j] + E[j-20,i-50]) & \mid \phi(\texttt{/i,j/,/51:60,21:50/}) \rightarrow \alpha_5 \\
SUM_{j=1:100:1}(H[i,j] + F[j-20,i-60]) & \mid \phi(\texttt{/i,j/,/61:100,21:60/}) \rightarrow \alpha_6 \\
SUM_{j=1:100:1}(H[i,j] + G[j-60,i-60]) & \mid \phi(\texttt{/i,j/,/61:100,61:100/}) \rightarrow \alpha_7
\end{cases}
$$

Step 2. This step is to find the sections which are with the same reduction pattern. According to the derived data access function in step 1, there are seven segmentation descriptors, $\alpha_1, \ldots, \alpha_7$. Also, the reduction is driven by the index variable j. Therefore, for $\alpha_1, \ldots$, and α_7, we remove the index domain related to j. See below:

$\alpha_1=\phi(/i,j/,/1:40,1:50/) \Rightarrow \alpha'_1=\phi(/i/,/1:40/)$.

$\alpha_2=\phi(/i,j/,/1:60,51:100/) \Rightarrow \alpha'_2=\phi(/i/,/1:60/)$.

$\alpha_3=\phi(/i,j/,/41:50,1:50/) \Rightarrow \alpha'_3=\phi(/i/,/41:50/)$.

$\alpha_4=\phi(/i,j/,/51:100,1:20/) \Rightarrow \alpha'_4=\phi(/i/,/51:100/)$.

$\alpha_5=\phi(/i,j/,/51:60,21:50/) \Rightarrow \alpha'_5=\phi(/i/,/51:60/)$.

$\alpha_6=\phi(/i,j/,/61:100,21:60/) \Rightarrow \alpha'_6=\phi(/i/,/61:100/)$.

$\alpha_7=\phi(/i,j/,/61:100,61:100/) \Rightarrow \alpha'_7=\phi(/i/,/61:100/)$.

Referring to *Fig. 3*, we can easily find that there are four sections where are with same reduction pattern. They are $\alpha'_1 \cap \alpha'_2=\phi(/i/,/1:40/)$, $\alpha'_2 \cap \alpha'_3 = \phi(/i/,/41:50/)$, $\alpha'_2 \cap \alpha'_4 \cap \alpha'_5 = \phi(/i/,/51:60/)$ and $\alpha'_4 \cap \alpha'_6 \cap \alpha'_7 = \phi(/i/,/61:100/)$. We call $\{\alpha'_1, \alpha'_2\}$, $\{\alpha'_2, \alpha'_3\}$, $\{\alpha'_2, \alpha'_4, \alpha'_5\}$, and $\{\alpha'_4, \alpha'_6, \alpha'_7\}$ are the *greatest-common-range sets* of $\{\alpha'_1, \alpha'_2, \alpha'_3\ \alpha'_4, \alpha'_5, \alpha'_6, \alpha'_7\}$. The method to derive the *greatest-common-range set* is formally described in Definition 4.

Step 3. Finally, for each greatest-common-range set, we use it to construct a data access pattern. We illustrate how to constrcut a data access pattern from $\{\alpha'_1, \alpha'_2\}$:

$E[i] = SUM_{j=1:100:1 \cap 1:50:1}(H[i,j] + A[j,i]) + SUM_{j=1:100:1 \cap 51:100:1}(H[i,j] + B[j-50,i]) \mid \phi(/i/,/1:40/) \cap \phi(/i/,/1:60/)$

$\Rightarrow E[i] = SUM_{j=1:50:1}(H[i,j] + A[j,i]) + SUM_{j=51:100:1}(H[i,j] + B[j-50,i] \mid \phi(/i/,/1:40/)$

Note that the reduction results from different regions have to be added together. Therefore, we sum the results of $SUM_{j=1:50:1}(H[i,j] + A[j,i])$ and $SUM_{j=51:100:1}(H[i,j] + B[j-50,i])$ in the above statement, together.

Here is the final synthesized data access function of our example:

$$
E[i] = \begin{cases}
SUM_{j=1:50:1}(H[i,j] + A[j,i]) + SUM_{j=51:100:1}(H[i,j] + B[j-50,i]) & \mid \phi(/i/,/1:40/) \\
SUM_{j=1:50:1}(H[i,j] + B[j-50,i]) + SUM_{j=51:100:1}(H[i,j] + C[j,i-40]) & \mid \phi(/i/,/41:50/) \\
SUM_{j=51:100:1}(H[i,j] + B[j-50,i]) + SUM_{j=1:20:1}(H[i,j] + D[j,i-50]) + SUM_{j=21:50:1}(H[i,j] + E[j-20,i-50]) & \mid \phi(/i/,/51:60/) \\
SUM_{j=1:20:1}(H[i,j] + D[j,i-50]) + SUM_{j=21:60:1}(H[i,j] + F[j-20,i-60]) + SUM_{j=61:100:1}(H[i,j] + G[j-60,i-60]) & \mid \phi(/i/,/61:100/)
\end{cases}
$$

Algorithm 4 describes the technique to handle the case above involved in synthesizing a multiple-clause data access function with a reduction operation. Definition 4 below first lists an auxiliary definition needed for *Algorithm 4* in grouping sections with the same reduction patterns together.

Definition 4 A group of segmentation descriptors may derive several greatest-common-range sets. The intersection of the elements in a greatest-common-range set is not empty set. Also, if M is the intersection of elements in a greatest-common-range set, then the intersection of another segmentation descriptors with M is an empty set. A group of segmentation descriptors $\{\ \alpha'_{a_1}, \alpha'_{a_2}, \cdots, \alpha'_{a_x}\ \}$ is a greatest-common-range set of $\{\alpha'_1, \alpha'_2, \cdots, \alpha'_n\ \}$, if

Algorithm 4 To synthesize a multiple-clause data access function and an extended data access function.
Step 1.

Synthesize the two data access functions as described in Section 7.2. Without loss of generality, assume we have the synthesized data access function as follows:

$$E[i_1, i_2, \cdots, i_{k-1}, i_{k+1}, \cdots, i_p] = \begin{cases} \bigoplus_{R_1}(DAP_1) & | \quad \alpha_1 \\ \bigoplus_{R_2}(DAP_2) & | \quad \alpha_2 \\ \cdots & \\ \bigoplus_{R_n}(DAP_n) & | \quad \alpha_n \end{cases}$$

The number of dimensions of array E is $p-1$. The number of dimensions of segmentation descriptor α_i is p. DAP_i denotes a data access pattern. $\bigoplus$ represents the reduction operation and R_i describes range of the reduction operation. Suppose R_1, R_2, $\cdots$, and R_n, are to reduce the k-th dimension of the array.

Step 2.

Let α'_i, $1 \leq i \leq n$, be derived from α_i by removing the k-th dimension of α_i. Find all of the **greatest-common-range set** of $\{\alpha'_1, \alpha'_2, \cdots, \alpha'_n\}$. The definition of greatest-common-range set is shown in Definition 4.

Step 3.

For each greatest-common-range set, we use it to construct a data access pattern. Assume $\{\alpha'_{a_1}, \alpha'_{a_2}, \cdots, \alpha'_{a_x}\}$ is a greatest-common-range set which we find in *Step 2*. According to this greatest-common-range set, we can obtain a data access pattern as follows:

$$E[i_1, i_2, \cdots, i_{k-1}, i_{k+1}, \cdots, i_p] = \begin{cases} \bigoplus(\bigoplus_{r_1}(DAP_{a1}), \bigoplus_{r_2}(DAP_{a2}), \cdots, \bigoplus_{r_x}(DAP_{ax})) & | \quad \Gamma \\ \cdots & \\ \cdots & \end{cases}$$

$$\Gamma = \alpha'_{a_1} \cap \alpha'_{a_2} \cap \cdots \cap \alpha'_{a_x}$$

where $\mathtt{r_i}$ = (index range of the k-th dimension of α_{a_i}) $\cap R_i$.
Finally, if there are N greatest-common-range sets, we will get a data access function with N data access patterns.

End Of Algorithm

(1)$\{\alpha'_{a_1}, \alpha'_{a_2}, \cdots, \alpha'_{a_x}\} \subseteq \{\alpha'_1, \alpha'_2, \cdots, \alpha'_n\}$.

(2) $\alpha'_{a_1} \cap \alpha'_{a_2} \cap \cdots \cap \alpha'_{a_x} \neq \emptyset$.

(3) $\alpha'_{a_1} \cap \alpha'_{a_2} \cap \cdots \cap \alpha'_{a_x} \cap \alpha'_b = \emptyset$, $\forall\ \alpha'_b \in \{\alpha'_1, \alpha'_2, \cdots, \alpha'_n\} - \{\alpha'_{a_1}, \alpha'_{a_2}, \cdots, \alpha'_{a_x}\}$.

Each greatest-common-range set represents a section which is with the same reduction pattern.