Simplification of Array Access Patterns for Compiler Optimizations *

Yunheung Paek‡ Jay Hoeflinger† David Padua†

‡ New Jersey Institute of Technology
paek@cis.njit.edu
† University of Illinois at Urbana-Champaign
{hoefling,padua}@uiuc.edu

Abstract

Existing array region representation techniques are sensitive to the complexity of array subscripts. In general, these techniques are very accurate and efficient for simple subscript expressions, but lose accuracy or require potentially expensive algorithms for complex subscripts. We found that in scientific applications, many access patterns are simple even when the subscript expressions are complex. In this work, we present a new, general array access representation and define operations for it. This allows us to aggregate and simplify the representation enough that precise region operations may be applied to enable compiler optimizations. Our experiments show that these techniques hold promise for speeding up applications.

1 Introduction

The array is one of the most important data structures in imperative programs. Particularly in numerical applications, almost all computation is performed on arrays. Therefore, the identification of the array elements accessed within a *program section* (e.g., a loop or a subroutine) by a particular reference (we call this *array access analysis*) is crucial to programming and compiler optimizations.

Figure 1 shows a reference to an m-dimensional array X with subscript function $(s_1(\mathbf{i}), \cdots, s_m(\mathbf{i}))$ defined on the set of indices $\mathbf{i} = (i_1, i_2, \cdots, i_d)$ within the program section $\mathcal{P}$ [†]. In array access analysis, the set of all elements of X accessed during the execution of section $\mathcal{P}$ (we call this the *access region* of X in $\mathcal{P}$) must be determined. For this purpose, when doing array access analysis for the references to an array, a compiler would first calculate a *Reference Descriptor* (RD) for each, which extends the array reference with range constraints. For instance, given that i_k ranges from l_k to u_k in $\mathcal{P}$ (for $k = 1, 2, \cdots, d$), the RD for the access in Figure 1 is

$$\text{“}X(s_1(\mathbf{i}), s_2(\mathbf{i}), \cdots, s_m(\mathbf{i})) \quad \textit{subject to} \\ l_k \le i_k \le u_k, \text{ for } k = 1, 2, \cdots, d.\text{”}$$

Next, the compiler would summarize all the RDs in the section and store their union in some standard representation [9, 15, 19]. Simple accesses can be summarized with simple representations without losing precision in access analysis. For example, in Figure 2, the region accessed by reference $b(i_1, i_2)$ can be represented by $b(0{:}n_b{:}1, 0{:}n_b{:}1)$ using the traditional *triplet notation.* In general, more powerful representations are required to accurately represent other accesses with more complex subscripts, such as the references to array a in Figure 2.

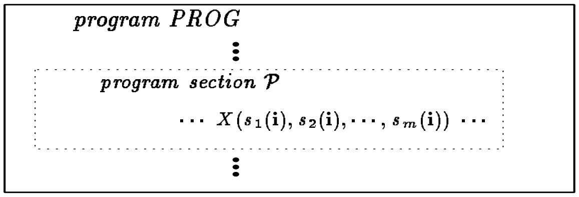
program PROG
⋮
program section $\mathcal{P}$
$\cdots\ X(s_1(\mathbf{i}), s_2(\mathbf{i}), \cdots, s_m(\mathbf{i}))\ \cdots$
⋮

Figure 1: Access to array X with indices $i_k, 1 \le k \le d$, in the program section $\mathcal{P}$ such that $l_k \le i_k \le u_k$.

```
program PROG
real $a[0:n_a]$, $b[0:n_b, 0:n_b]$
 ...
for $i_1 = 0$ to $n_b$ with step 1 do
    $b(i_1, i_1) = \cdots$
    for $i_2 = 0$ to $n_b$ with step 1 do
        $b(i_1, i_2) = a(8*i_1 + i_2) + a(i_1 + 8*i_2 + l)$
    end
    for $i_3 = 0$ to $i_1$ with step 1 do
        $a(i_1*(i_1+1)/2 + i_3) = \cdots$
    end
end
 ...
```

Figure 2: Code example where the section $\mathcal{P}$ is the i_1-loop

[*] This work is supported in part by Army contract DABT63-95-C-0097; Army contract N66001-97-C-8532; NSF contract MIP-9619351; and a Partnership Award from IBM. This work is not necessarily representative of the positions of the Army or the Government.

[†] Indices are all basic induction variables of loops surrounding the reference

To appear in the Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation, Montreal, Canada.

Conventional wisdom holds that complex array subscripts result in complex access patterns. However, we believe that complex subscripts often form simple access patterns. This belief is based on informal observations of many programs. For instance, although the subscript function for a in the i_3-loop seems complex, we can see in Figure 3 that the actual access pattern is simple. In many cases, the simplicity of the real access is hidden inside the subscript expressions, making it difficult to discover. Sometimes originally simple subscript expressions are converted to complex ones during compiler transformations, such as induction variable substitution, value propagation [3], and subroutine inlining [18, 13], although the original access patterns remain intact. Previous techniques sometimes fail to recognize these simple patterns and, as a result, lose accuracy in their access analysis.

When an m-dimensional array is allocated in memory, it is linearized and laid out in either *row-major* or *column-major* order, depending on the language being used. Thus, we view accessing an array as traversing a linear memory space. Figure 3 shows how the two arrays a and b are mapped to memory and accessed within the i_1-loop of Figure 2.

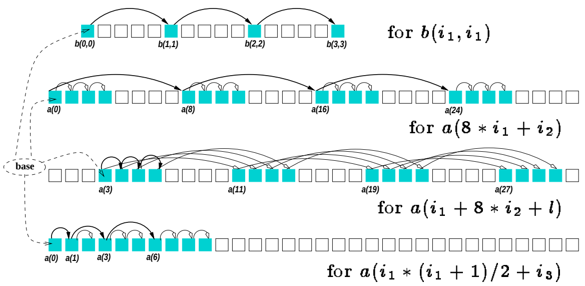

Figure 3: Access patterns in the i_1-loop in Figure 2 for $n_b = l = 3$: gray boxes represent the array elements accessed, and arrows with black heads and white heads keep track of the access driven by indices i_1 and i_2(or i_3), respectively.

1.1 Access Descriptor Exploiting Regularity of Memory Accesses

For the most part, the simplicity of accesses is exposed via a *regularity* of access structure within limited sections of a program. By regularity we mean that the same access patterns are repeated in the traversal of memory. The patterns are characterized by two factors: the **stride** and the **span**.

The stride records the distance traveled in the memory space when an index is incremented. For example, for array b of Figure 3, it can be seen that the incrementing of index i_1 causes movement through memory in strides of size $n_b + 2$, which actually corresponds to accessing the main diagonal of b. In the two references to a within the i_2-loop, index i_1 causes a stride of 8 in the first and 1 in the second, while index i_2 causes a stride of 1 in the first and 8 in the second.

The span records the total distance traveled in the memory space due to a single index with the other indices fixed (that is, the difference between the last offset and the first offset of the array elements accessed). Consider the reference $a(8 * i_1 + i_2)$. Notice that when the index i_1 independently iterates through its entire range of values with the value of i_2 fixed, the range of the access within memory has length $8n_b$. Similarly, when i_2 iterates alone, the access range has length n_b.

The memory traversal due to the independent iteration of each index is characterized by its strides and spans, as can be seen from Figure 3. We pair the stride and span produced by a single index for an array reference to describe the access pattern created by that index. Given a set of indices $(i_1, i_2, \cdots, i_d)$, the collection of the stride/span pairs for all indices represents the entire access pattern for array A, denoted by

$$\mathcal{A}^{\delta_{i_1},\delta_{i_2},\cdots,\delta_{i_d}}_{\sigma_{i_1},\sigma_{i_2},\cdots,\sigma_{i_d}} + \tau$$

where δ_{i_k} and σ_{i_k} are the stride and span due to index i_k, and τ is the **base offset**, the offset of the first access from the beginning of the array. We call this form a *Linear Memory Access Descriptor* (LMAD) and each stride/span pair a **dimension** of the descriptor. Also, we call index i_k, which is associated with the kth dimension $(\delta_{i_k}, \sigma_{i_k})$, the **$k$th dimension index** of the descriptor.

Using the LMAD form, we can summarize the four accesses in Figure 3 with a 1-dimensional descriptor $\mathcal{B}^{n_b+2}_{n_b(n_b+2)}+0$ for $b(i_1,i_1)$, and 2-dimensional descriptors

$$\mathcal{A}^{8,1}_{8n_b,n_b}+0,\ \mathcal{A}^{1,8}_{n_b,8n_b}+l \text{ and } \mathcal{A}^{i_1+1,1}_{\frac{n_b(n_b+1)}{2},i_1}+0$$

for $a(8{*}i_1{+}i_2)$, $a(i_1{+}8{*}i_2{+}l)$ and $a(i_1{*}(i_1{+}1)/2{+}i_3)$, respectively.

Definition 1 *On the assumption that two LMADs* $\mathbf{A}$ *and* $\mathbf{A}'$ *represent the access regions* $\mathcal{R}$ *and* $\mathcal{R}'$*, respectively,*

1. $\mathbf{A} \cup \mathbf{A}'$ *represents the* **aggregated** *LMAD of the two access regions, that is,* $\mathcal{R} \cup \mathcal{R}'$.
2. *If* $\mathcal{R}'$ *is a subregion of* $\mathcal{R}$ *(that is,* $\mathcal{R}' \subset \mathcal{R}$*), then we write* $\mathbf{A}' \subset \mathbf{A}$.
3. *Let* $\mathbf{A} = \mathcal{A}^{\delta_1,\cdots,\delta_k,\cdots,\delta_d}_{\sigma_1,\cdots,\sigma_k,\cdots,\sigma_d} + \tau$. *Suppose* $\mathbf{A}'$ *is built by eliminating the* k*th dimension* (δ_k, σ_k) *from* $\mathbf{A}$*, that is,* $\mathcal{A}^{\delta_1,\cdots,\delta_{k-1},\delta_{k+1}\cdots,\delta_d}_{\sigma_1,\cdots,\sigma_{k-1},\sigma_{k+1},\cdots,\sigma_d} + \tau$. *We call* $\mathbf{A}'$ *the* k-**subLMAD** *of* $\mathbf{A}$.

Notice that when $\mathbf{A}'$ (with access region $\mathcal{R}'$) is the k-subLMAD of $\mathbf{A}$ (with access region $\mathcal{R}$), then $\mathcal{R}' \subset \mathcal{R}$.

1.2 Analysis of Subscripting Patterns

The development of the access region notation was originally motivated by the project [24, 25] to retarget the *Polaris* compiler [4] at distributed memory multiprocessors. In that project, the triplet notation used by Polaris for array access analysis prevented us from generating efficient code for our target multiprocessors because subscript expressions that could not be represented accurately in triplet notation limited our compiler optimizations. The eventual success in that project was due in part to our use of the LMAD notation. Section 6 presents a few results on the improvements due to use of the LMAD.

To develop a new access region notation, we needed to better understand the actual access patterns in real applications. For this, we chose fourteen Fortran77 programs, including codes from the SPEC95fp and Perfect

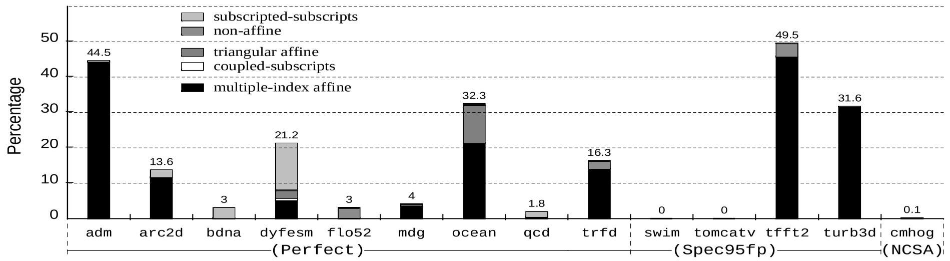

Figure 4: Percentage of non-triplet-representable access summaries versus total number of access summaries

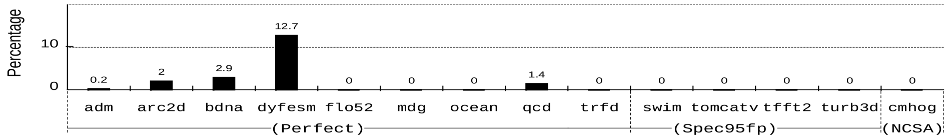

Figure 5: Percentage of access summaries which are not provably-monotonic versus total number of access summaries

benchmarks, and one from a set of production codes obtained from the National Center for Supercomputing Applications (NCSA) to study. After applying interprocedural value propagation, induction variable substitution, and forward substitution within these codes, we summarized each array reference to each of its enclosing do loops, counted how many of these summaries would not be representable in triplet notation, and plotted their percentage with respect to the total number of summaries in Figure 4. For this analysis, we divided the array accesses that could not be represented by triplet notation into the following five categories:

subscripted-subscripts : accesses due to references with subscripted-subscript expressions;

non-affine : accesses due to references with non-affine subscript expressions;

triangular affine : accesses due to references within a triangular loop;

coupled-subscripts : accesses due to references with coupled-subscript expressions [20];

multiple index affine : accesses due to references containing multiple indices in a subscript position.

In this classification, each category excludes those above it. For instance, a reference with a subscripted-subscript inside a triangular loop would be counted as subscripted-subscript, and not triangular affine.

In order for the difference between the last offset and the first offset (the span) to represent the true distance moved for a dimension, the subscript function must cause movement to be consistently in the same direction. Such a function is called *monotonic* [3], which will be formally defined in Section 4. This implies that the LMAD can be accurate only when the subscripting functions are monotonic. Thus, to see how often the LMAD can be accurate in reality, we determined the percentage of array accesses that were provably monotonic at compile time.

By their nature, all categories of references except subscripted-subscripts and non-affine are monotonic. But we checked all non-affine references in our set of test codes and, unexpectedly, all of those accesses were provably monotonic. Only the subscripted-subscripts were not provably monotonic at compile time. This data is presented in Figure 5.

From these results, we learned that most subscript functions encountered in the programs we tested are monotonic. This indicates the general tendency that for the iteration of a single index in a program section, an array is accessed in one direction either from a low to a high address in memory or vice versa. We also conclude that use of the LMAD makes possible an improvement in the accuracy of the representation over that obtained with triplet notation.

The purpose of this paper is to show how the LMAD is used to analyze and simplify array access patterns in a program for more efficient and accurate compiler optimizations.

1.3 Organization of this Paper

Section 2 discusses previous work done on array access analysis and representation. Section 3 discusses several properties of LMADs and classifies access patterns that allow simplification of LMADs. Section 4 shows how to build a LMAD from a RD. Section 5 describes basic principles and methods to handle the access patterns classified in Section 3. Section 6 shows examples encountered in our experiments where LMADs were advantageous to our optimizing compiler, along with some performance results. We briefly discuss the impact of this work on our current compiler project in Section 7, and present our conclusions in Section 8.

2 Array Access Analysis Techniques

Work on representing array access regions has followed three major approaches: triplet-notation-based, reference-list-based, and linear constraint-based.

2.1 Triplet-notation-based techniques

Triplet notation is an array region representation for a set of integer values that start at a lower bound and proceed to an upper bound via a stride for each array dimension declared by the programmer. For instance, the notation for a 3-dimensional array A is given by

$$A(lb_1{:}ub_1{:}s_1,\ lb_2{:}ub_2{:}s_2,\ lb_3{:}ub_3{:}s_3)$$

where lb_k is the kth lower bound, ub_k is the kth upper bound, and s_k is the kth stride. Triplet notation is simple, yet practical. Typically, the *region operations* (e.g., union, subtraction, and intersection) defined on the notation can be implemented with fast linear algorithms. The study of Shen et al [28] indicated that most real-world access patterns in scientific programs are representable by triplet notation. This is, in fact, the reason why many researchers, including several in our own research group at Illinois, have used the notation to implement their compiler techniques (including array privatization, dependence analysis, and message generation [3, 7, 16, 31]).

However, as discussed in Section 1.2, the limited expressive power of the triplet notation often hinders analysis in some important cases. To alleviate this problem, researchers at Rice University [6] have devised several variants of *regular section descriptors* (RSDs), with operations defined on a lattice. RSDs are able to express single array elements, complete rows and columns, and diagonals. *Restricted* RSDs [15] were devised to handle coupled subscripts, and then *Bounded* RSDs were devised to further improve the accuracy with symbolic bound information. Researchers at the University of Minnesota have used Guarded Array Regions [14], which are equivalent to Bounded RSDs with an additional predicate (guard). More information can be added to the guard to sharpen the accuracy in a given situation.

2.2 Reference-list based techniques

Li and Yew proposed a reference-list based representation, called an *atom image* [19], which captures the coefficients of the loop indices and the loop bounds of each surrounding loop. Burke and Cytron represented array references in one-dimensional form by *linearization* [5]. These reference-list based techniques lose no precision for any array access because they rely on making a list of each individual array reference in a program section. They are meant to capture program information, but not summarize it.

2.3 Linear-constraint based techniques

In linear constraint-based techniques, the set of linear constraints is constructed from the subscripting function, loop bounds, and other information found in the program, similar to our calculation of the RD. The following is an example:

$$\left\{ A(x_1, x_2) \;\middle|\; \begin{array}{c} l_1 \le x_1 \le u_1 \\ l_2 \le x_2 \le u_2 \\ l_3 \le x_1 + x_2 \le u_3 \\ l_4 \le x_1 - x_2 \le u_4 \end{array} \right\}$$

With these techniques, array accesses can be expressed as *convex regions* in a geometrical space. The linear constraint-based techniques that were first proposed by Triolet, et al [30] have been widely used as an alternative way to summarize array accesses. In particular, these techniques have been used for dependence analysis. When a potential dependence between two array references is being tested, the linear inequalities associated with the two references are aggregated to form a linear system and the feasibility of the system is tested using Fourier-Motzkin elimination [11] techniques. The Omega Test [26] is an example of a dependence test built in this way. The PIPS project at Ècole des Mines de Paris [9] has added an indicator of the accuracy of the representation, referred to as *MUST/MAY*, to the representation itself. The notion of MUST/MAY approximations helps a compiler to determine when a result is accurate or inaccurate.

Linear constraint-based techniques are generally considered more precise than triplet notation in handling access patterns with non-rectangular expressions (see Figure 4). However, they also have several critical drawbacks. First, the Fourier-Motzkin linear system solver requires worst-case exponential time algorithms [2]. Balasundaram and Kennedy [1] proposed a simplified form of linear constraint representation, called *simple sections*, that eliminates the need for such expensive algorithms, but at the cost of accuracy. Although simple sections can represent many of the commonly occurring forms, such as a whole array, a single row/column, a diagonal, or a triangular section, they are limited in that they cannot express the whole range of constraints found in programs. Second, Fourier-Motzkin is limited to affine expressions. To overcome this limitation, Pugh and Wonnacott [27] have developed techniques for replacing non-affine terms occurring in array subscripts with *uninterpreted function symbols*, but this does not handle all situations involving non-affine terms. Third, Fourier-Motzkin requires that the linear inequalities form a convex hull, forcing a loss of accuracy when regions must be altered to maintain the convex form.

Work on the SUIF system at Stanford uses a representation [21] very similar to that of PIPS. SUIF uses a set of region operations for systems of linear inequalities and special algorithms for maintaining the convex shape of the regions during the analysis.

To recap, the linear constraint-based representation improves accuracy over triplet notation, but still loses accuracy and requires potentially expensive algorithms for many complex subscripting expressions (the frequencies of which are shown in Figure 4). The performance of a compiler based on linear constraints would be improved if those expressions could be simplified.

3 The Similarity of Array Access Patterns

In addition to the property of regularity discussed in Section 1, different accesses in the same program section often have a *similarity* of pattern because multiple references to an array within the section are generally accessed using the same indices and similar subscript expressions. Figure 3 shows that the accesses represented by descriptors $\mathcal{A}^{8,1}_{24,3} + 0$ and $\mathcal{A}^{1,8}_{3,24} + l$ describe exactly the same memory access pattern but with different base offsets. This example helps demonstrate that the order of the stride/span pairs on the LMAD does not affect the access pattern represented by it. We call such similar access patterns *isomorphic*.

Definition 2 *Let* $\mathbf{A}$ *and* $\mathbf{A}'$ *be two LMADs. If* $\mathbf{A}'$ *has the same stride/span pairs as* $\mathbf{A}$*, regardless of their order, then* $\mathbf{A}$ *and* $\mathbf{A}'$ *are* **isomorphic***, denoted by* $\mathbf{A}//\mathbf{A}'$*, meaning the access regions described have the same shape and structure.*

Definition 3 *Two LMADs* $\mathbf{A}$ *and* $\mathbf{A}'$ *are said to be* **equivalent***, denoted by* $\mathbf{A} \equiv \mathbf{A}'$*, if they represent the same access region.*

If two LMADs are isomorphic and have the same base offset, they are equivalent. For instance, since the two descriptors $\mathcal{A}^{8,1}_{24,3} + 0$ and $\mathcal{A}^{1,8}_{3,24} + l$ are isomorphic, they would be equivalent if we could show that $l = 0$.

To illustrate another type of similarity, consider the loop in Figure 6. A descriptor $\mathcal{A}^{3,1}_{9,2} + 1$, whose accesses are denoted by dashed lines in the figure, represents the access region produced by $a(i{+}j{+}1)$. Notice that another descriptor $\mathcal{A}^{1,2}_{1,10} + 1$, whose accesses are denoted by the lower solid lines, represents the same region as the original descriptor. Even a 1-dimensional descriptor can represent the same access, such as $\mathcal{A}^{1}_{11} + 1$, whose accesses are denoted by the upper solid lines.

```
real $a(0:n)$
...
for $j = 0$ to $2$ with step $1$ do
   for $i = 0$ to $9$ with step $3$ do
      $a(i + j + 1) = \cdots$
   end
end
```

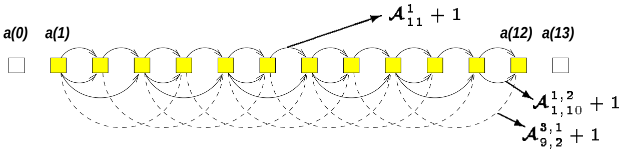
$\mathcal{A}^{1}_{11} + 1$
a(0)
a(1)
a(12)
a(13)
$\mathcal{A}^{1,2}_{1,10} + 1$
$\mathcal{A}^{3,1}_{9,2} + 1$

Figure 6: Equivalent regions for the accesses to array a, which cover the region from $a(1)$ to $a(12)$.

All three descriptors in Figure 6 are equivalent. From these examples, we learn that an access region can be represented by numerous LMADs consisting of different stride/span pairs. In principle, we can show that the same access region can be represented with an infinite number of equivalent LMADs by Theorem 1, which will be used in Section 5.

Theorem 1 *A LMAD* $\mathcal{A}^{\sigma_1,\cdots,\sigma_d}_{\delta_1,\cdots,\delta_d} + \tau$ *can be expanded to form other equivalent descriptors by adding a dimension* $(\delta^*, 0)$ *in any position, such as* $\mathcal{A}^{\delta_1,\cdots,\delta^*,\cdots,\delta_d}_{\sigma_1,\cdots,0,\cdots,\sigma_d} + \tau$*, where* δ^* *can be any integer.*

PROOF: A dimension describes the movement from a lower bound to an upper bound with a stride. The span is defined as the difference between the upper bound and the lower bound. If the span is zero, the upper bound and lower bound are the same, thus describing no movement at all. A dimension involving no movement can have any stride, and still neither adds nor subtracts elements to/from a given access region. ■

As we have mentioned, the complexity of subscript expressions sometimes prevents the representation techniques described in Section 2 from accurately representing an access. However, if the access patterns are represented in a sufficiently general form, they often can be aggregated and simplified into forms which are representable in the notation of choice, by using techniques such as those discussed in this paper. For example, a direct translation to triplet notation of the access in the loop of Figure 6 would not be possible because the access involves multiple index subscripts (see Figure 4). However, using our algorithms, we can show that the original pattern is equivalent to an access with $\mathcal{A}^{1}_{11} + 1$; therefore, we can use the triplet notation $a(1{:}12{:}1)$ to accurately represent the access.

Our work is based on the observation that typical scientific programs have several common forms of access patterns produced by subscript expressions, regardless of their complexity. We call these forms *coalesceable*, *interleaved*, and *contiguous*. These are useful for our purposes because, whenever one of these forms appears, the original access pattern can be transformed into a simpler one.

Coalesceable Accesses

Given an array access represented with a LMAD $\mathbf{A}$, we call the access *coalesceable* if it also can be represented with another LMAD $\mathbf{A}'$ equivalent to $\mathbf{A}$, but with fewer dimensions than $\mathbf{A}$. One typical example of a coalesceable access is one that moves with a small stride due to one index and due to a different index, strides over the accesses of the first stride to the very next element in the sequence. For instance, the reference $a(i_1{*}(i_1{+}1)/2{+}i_3)$ in Figure 2 accesses $i_1 + 1$ consecutive elements with stride 1 for every iteration of the i_1-loop, then jumps over those elements with stride $i_1 + 1$ to the next element. In a case like this, we can show that the number of dimensions can be reduced by one. We can show that

$$(\mathcal{A}^{1,i_1+1}_{i_1,\frac{n_b(n_b+1)}{2}} + 0) \equiv (\mathcal{A}^{1}_{\frac{n_b(n_b+1)}{2}+n_b} + 0),$$

using the algorithm presented in Figure 10.

Interleaved Accesses

We call array accesses *interleaved* when their dimensions have the same strides and they are offset from each other by a fixed distance which divides one of their strides. An example is shown in Figure 7, where we can see that the access patterns for arrays x and y have this property. We can see that the union of the three access regions for x is equal to the whole region from $x(l)$ to $x(u+4)$ with stride 2, which can be represented by a single descriptor $\mathcal{X}^{2}_{u-l+4} + l$. This implies

$$(\mathcal{X}^{6}_{u-l} + l) \cup (\mathcal{X}^{6}_{u-l} + l + 2) \cup (\mathcal{X}^{6}_{u-l} + l + 4)$$
$$\equiv (\mathcal{X}^{2}_{u-l+4} + l).$$

Contiguous Accesses

We call LMADs *contiguous* when the access patterns they represent are similar, can be fit together to cover a portion of the array without a break, and can be expressed in a single LMAD. Figure 8 illustrates four access patterns for array a. From this example, we can clearly see the similarity between those accesses; and we see that when viewed from outside the outermost

```
real $x(0:n)$, $y(0:m)$
...
for $i = l$ to $u$ with step 6 do
    $temp = temp + x(i) * y(i)$
                $+x(i+2) * y(i+2)$
                $+x(i+4) * y(i+4)$
end
```

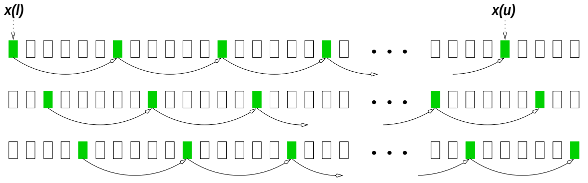

Figure 7: Code example of the dot product of two vectors x and y with stride 2, and the illustration of the access patterns for x in memory, represented by three descriptors $\mathcal{X}^{6}_{u-l}+l$, $\mathcal{X}^{6}_{u-l}+l+2$, and $\mathcal{X}^{6}_{u-l}+l+4$, respectively

loop, they fit together to form an unbroken stream of access. This results in

$$(\mathcal{A}^{5}_{15}+1) \cup (\mathcal{A}^{5,1}_{15,1}+2) \cup (\mathcal{A}^{1,5}_{3,15}+3) \cup (\mathcal{A}^{5}_{15}+4)$$
$$\equiv (\mathcal{A}^{1}_{19}+1)$$

which indicates that the union of the four access regions can be represented with a single LMAD.

```
real $a(0:n)$
...
for $i$ = 0 to 3 with step 1 do
    $a(5*i+1) = \cdots$
    for $k$ = 1 to 2 with step 1 do
        $a(k+5*i+1) = \cdots$
    end
    for $j$ = 1 to 4 with step 1 do
        $a(5*i+j+2) = a(5*i+4) + \cdots$
    end
end
```

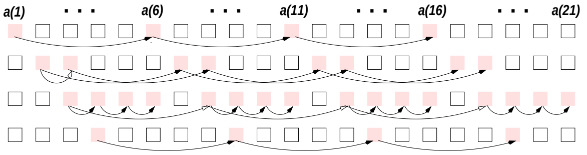

Figure 8: Similar access patterns with four descriptors $\mathcal{A}^{5}_{15}+1$, $\mathcal{A}^{5,1}_{15,1}+2$, $\mathcal{A}^{1,5}_{3,15}+3$, and $\mathcal{A}^{5}_{15}+4$

4 Generating Linear Memory Access Descriptors

Let $\mathbf{Z}$ denote the set of all integers and $\mathbf{Z}^m$ the set of all m-tuples of integers. We assume here, for simplicity of explanation, that all subscripts of an array start at 0, and all loops are normalized with a stride of 1. For example, the array X in Figure 1 is declared as $X(0{:}n_1, 0{:}n_2, \cdots, 0{:}n_m)$. We define the *array space* of X, $\mathbf{Z}^m_X$, as the set of m-tuples:

$$\mathbf{Z}^m_X = \{(r_1, r_2, \cdots, r_m) \in \mathbf{Z}^m | 0 \le r_k \le n_k, 1 \le k \le m\}.$$

Because vector $\mathbf{r} = (r_1, r_2, \cdots, r_m)$ represents the address in the array space of X, to access the actual data stored in memory it must be mapped to a single integer that is the offset from the beginning of the array. We define this mapping, denoted by $F_X : \mathbf{Z}^m_X \rightarrow \mathbf{Z}$, as

$$F_X(\mathbf{r}) = F_X(r_1, r_2, \cdots, r_m) = \sum_{k=1}^{m} r_k \cdot \lambda_k$$

where, provided X is allocated in column-major order, $\lambda_1 = 1$ and $\lambda_k = \lambda_{k-1} \cdot (n_{k-1}+1)$ for $k \neq 1$. If the array is allocated in row-major order, $\lambda_m = 1$ and $\lambda_k = \lambda_{k+1} \cdot (n_{k+1}+1)$ for $k \neq m$. Applying the definition of F_X to the subscript function in Figure 1, we have a linearized form:

$$F_X(\mathbf{s}(\mathbf{i})) = s_1(\mathbf{i})\lambda_1 + s_2(\mathbf{i})\lambda_2 + \cdots + s_m(\mathbf{i})\lambda_m \qquad (1)$$

where we assume that $\mathbf{s}(\mathbf{i}) = (s_1(\mathbf{i}), s_2(\mathbf{i}), \cdots, s_m(\mathbf{i}))$.

Now, we show how we could use Equation 1 to calculate the dimensions (stride/span pairs) and base offset of the accesses made by an array reference $X(\mathbf{s}(\mathbf{i}))$. Let $\hat{\delta}^{h}_{i_k}$ denote the difference in $F_X(\mathbf{s}(\mathbf{i}))$ made by replacing i_k with $i_k + h$, where h is a positive integer. For general subscript functions, it can be represented by the *difference* operator Δ, defined by

$$\hat{\delta}^{h}_{i_k} = \frac{\Delta F_X(\mathbf{s}(\mathbf{i}))}{\Delta i_k} = \qquad (2)$$
$$\frac{F_X(\mathbf{s}(i_1, \cdots, i_k + h, \cdots, i_d)) - F_X(\mathbf{s}(i_1, \cdots, i_k, \cdots, i_d))}{h}$$

which is used to define a monotonic subscript function mentioned in Section 1.

Definition 4 *Let* $\mathbf{s}(\mathbf{i})$ *be a subscript function defined on* $\mathbf{i} = (i_1, i_2, \cdots, i_d)$. *For all the values of index* i_k *in the range between* l_k *and* u_k, *we say the function* $\mathbf{s}(\mathbf{i})$ *is* **nondecreasing** *for the index* i_k *if* $\hat{\delta}^{h}_{i_k} \ge 0$; *otherwise, we say* $\mathbf{s}(\mathbf{i})$ *is* **nonincreasing**. *The subscript function* $\mathbf{s}(\mathbf{i})$ *is* **monotonic** *for index* i_k *if* $\mathbf{s}(\mathbf{i})$ *is either nondecreasing or nonincreasing for* i_k.

Suppose $\mathbf{s}(\mathbf{i})$ is monotonic for all indices in $\mathbf{i}$. Then, the stride δ_{i_k} caused by the iteration of index $i_k \in \mathbf{i}$ can be obtained directly from $\hat{\delta}^{h}_{i_k}$, as follows:

$$\delta_{i_k} = \left|\hat{\delta}^{1}_{i_k}\right| =$$
$$\left|\sum_{j=1}^{m}(s_j(i_1, \cdots, i_k+1, \cdots, i_d) - s_j(i_1, \cdots, i_k, \cdots, i_d))\lambda_j\right|$$

where h is set to 1, as it is in *finite calculus* [12]. As stated in Section 1, the span σ_{i_k} is the distance moved in the memory region accessed during the iteration of i_k from l_k to u_k. If the function is monotonic, then, by finite calculus, this can be calculated by subtracting the function values of the two end-points of the intervals. Thus, the span is

$$\sigma_{i_k} = \left|\sum\nolimits_{l_k}^{u_k} \hat{\delta}^{1}_{i_k} \Delta i_k\right| =$$
$$\left|\sum_{j=1}^{m}(s_j(i_1, \cdots, u_k, \cdots, i_d) - s_j(i_1, \cdots, l_k, \cdots, i_d))\lambda_j\right|$$

where h is again 1. The sign of $\hat{\delta}^{1}_{i}$ denotes the direction of the movement through memory due to i in an array access. In our formulas, however, we ignore the sign by taking the absolute value, $|\hat{\delta}_i^1|$, for the stride δ_i, and changing the base offset because, as we stated in Section 1, the access pattern (or access region) is characterized by the stride size of the movement, not the direction. To show this with an example, consider Figure 9. Here, we have two accesses to array a, where $\hat{\delta}_{i_1}^1$ in the first is -1 (causing movement to the left with stride 1, starting from $a(4)$) and in the second is 1 (causing movement to the right with stride 1, starting from $a(2)$). However, despite the different access structures, we see that the access region (represented with gray boxes) for both accesses is actually identical.

```
for $i_2$ = 0 to 6 do
    for $i_1$ = 0 to 2 do
        $a(-i_1 + 4 * i_2 + 4) = \cdots$
        $a(i_1 + 4 * i_2 + 2) = \cdots$
    end
end
```

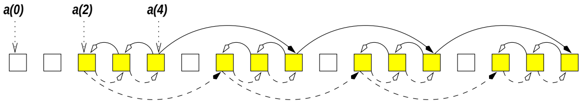

Figure 9: Access to array a through two references and their access patterns in memory: solid lines denote the access for $a(\text{-}i_1+4*i_2+4)$ and dashed lines the access for $a(i_1+4*i_2+2)$: arrows with white heads and black heads keep track of the access driven by indices i_1 and i_2, respectively.

As can be seen in Figure 9, the direction of access only adds to the complexity in access analysis. Thus, we remove the extra complexity by normalizing LMADs to a form in which all the directions of access are positive. This normalization requires us to change the base offset of the access to the lower bound of the access region. For example, in Figure 9, the original base of the access for $a(-i_1 + 4 * i_2 + 4)$ was $a(4)$; but, during normalization, the base must be changed to $a(2)$. For the normalized form, using Equations 1 and 2, we calculate the base offset of the access made by the reference $X(\mathbf{s}(\mathbf{i}))$ as follows:

$$\tau = \sum_{j=1}^{m} s_j(c_1, c_2, \cdots, c_d)\lambda_j$$

$$\text{where } c_k = \begin{cases} u_k & : \ \hat{\delta}_k^1 < 0 \\ l_k & : \ \hat{\delta}_k^1 \geq 0 \end{cases} \quad \text{for } 1 \leq k \leq d.$$

Note that τ is the minimum value of $F_X(\mathbf{s}(\mathbf{i}))$. The LMADs for the accesses in Figure 9 produced by our formulas would both be $\mathcal{A}_{2,12}^{1,4} + 2$. Thus, we can prove that the two references access the same array region.

5 Linear Memory Access Descriptor Manipulation

In this section, we discuss the techniques that identify the regularity and similarity of the three categories of access patterns discussed earlier and that use the characteristics to simplify or to aggregate their descriptors.

5.1 Coalesceable Accesses

Given a single LMAD, the algorithm in Figure 10 determines if the access represented by the LMAD is coalesceable. For example, the access for $a(i_1 * (i_1 + 1)/2 + i_3)$ in Figure 3 is represented by the descriptor

$$\mathcal{A}_{i_1, n_b(n_b+1)/2}^{1, i_1+1} + 0$$

which has two dimensions: $(1, i_1)$ due to index i_3 and $(i_1 + 1, \frac{n_b(n_b+1)}{2})$ due to i_1. These pairs are coalesceable because $\delta_{i_3}(= 1)$ divides $\delta_{i_1}(= i_1 + 1)$, and $\delta_{i_1} = \sigma_{i_3} + \delta_{i_3}$. Thus, we eliminate $(i_1 + 1, \frac{n_b(n_b+1)}{2})$ from the descriptor and update the original span $\sigma_{i_3}(= i_1)$ with $\frac{n_b(n_b+1)}{2} + i_1$. Since the new span contains i_1, we replace the index with its upper bound n_b, resulting in the 1-dimensional descriptor

$$\mathcal{A}_{n_b(n_b+1)/2+n_b+1}^{1} + 0.$$

This eliminates one dimension of the LMAD, thereby simplifying the representation of the original access pattern. In a similar way, we can show that the access for the reference $b(i_1, i_2)$ due to the two indices i_1 and i_2 in Figure 2 can be represented by the 1-dimensional descriptor

$$\mathcal{A}_{(n_b+1)^2-1}^{1} + 0.$$

```
                    Algorithm Coalesce

Input 1: dimension index set $\mathcal{I} = \{i_1, i_2 \cdots, i_d\}$
        with constraints $l_k \leq i_k \leq u_k$   for $k = 1, 2, \cdots, d$
Input 2:  LMAD $\mathbf{A} = \mathcal{A}_{\sigma_{i_1}, \sigma_{i_2}, \cdots, \sigma_{i_d}}^{\delta_{i_1}, \delta_{i_2}, \cdots, \delta_{i_d}} + \tau$
Note: Here $f[j \leftarrow x]$ means to substitute $x$ for $j$ in function $f$

Algorithm:
    while an unselected index pair $(i_j, i_k)$ from $\mathcal{I}$ remains do
        Select two dimensions $(\delta_{i_j}, \sigma_{i_j})$ and $(\delta_{i_k}, \sigma_{i_k})$ from $\mathbf{A}$;
        if $i_j$ or $i_k$ appears in the other dimensions of $\mathbf{A}$ then
            continue;
        fi
        if $\delta_{i_j}$ divides $\delta_{i_k}$ and $\delta_{i_k} \leq \sigma_{i_j} + \delta_{i_j}$ then
            if $i_k$ appears in $\sigma_{i_j}$ then
                if $\exists \hat{\imath}$, $l_k \leq \hat{\imath} < u_k$, such that
                    $\delta_{i_k} < \sigma_{i_j}[i_k \leftarrow \hat{\imath}] - \sigma_{i_j}[i_k \leftarrow \hat{\imath} + 1]$ then
                    continue;
                fi
                $\sigma_{i_j} = \sigma_{i_j}[i_k \leftarrow u_k]$;
            fi
            $\sigma_{i_j} \leftarrow \sigma_{i_j} + \sigma_{i_k}$;
            Eliminate $(\delta_{i_k}, \sigma_{i_k})$ from $\mathbf{A}$;
            $\mathcal{I} \leftarrow \mathcal{I} - \{i_k\}$;
        fi
    od
end
```

Figure 10: An algorithm that detects coalesceable accesses from a LMAD by comparing its stride/span pairs.

Algorithm *coalesce* does $O(d^2)$ dimension comparisons for a d-dimensional LMAD in the worst case.

5.2 Interleaved Accesses

In Figure 6, we can find that the access from $a(1)$ to $a(12)$ with stride 1 (represented by $\mathcal{A}_{11}^1 + 1$) comprises two separate interleaved accesses with stride 2 (represented by $\mathcal{A}_{10}^2 + 1$ and $\mathcal{A}_{10}^2 + 2$) or three accesses with stride 3 (represented by $\mathcal{A}_9^3+1$, $\mathcal{A}_9^3+2$, and $\mathcal{A}_9^3+3$). Theorem 2 shows how simplification can be applied to aggregate multiple regions with an interleaved structure; it gives us the flexibility to convert a single LMAD to its n-interleaved descriptors, or vice versa.

To determine whether n given d-dimensional descriptors are n-interleaved with respect to a given stride δ, we would first sort the dimensions of each by stride, an $O(nd \log d)$ process, then use a linear-time bucket sort to order the descriptors by base offsets, checking that in sorted order the offsets differ by the same δ/n. The complexity of this process is dominated by $O(nd \log d)$.

Theorem 2 *Let* $\mathbf{A} = \mathcal{A}^{\delta_1,\delta_2,\cdots,\delta_d}_{\sigma_1,\sigma_2,\cdots,\sigma_d} + \tau$. *For a dimension* (δ_k,σ_k) *in* $\mathbf{A}$, *where* δ_k *and* σ_k *are invariant, and a chosen* n, $1 \leq n \leq 1 + \frac{\sigma_k}{\delta_k}$, *there always exists a set of* n *descriptors*

$$\{\mathbf{A}_j \mid 1 \leq j \leq n\} = \left\{ \mathcal{A}^{\delta_1,\cdots,\delta'_k,\cdots,\delta_d}_{\sigma_1,\cdots,\sigma'_k,\cdots,\sigma_d} + \tau + (j-1)\delta_k \right\}$$

where $\delta'_k = n\delta_k$ *and* $\sigma'_k = \left\lfloor \frac{\sigma_k - (j-1)\delta_k}{\delta'_k} \right\rfloor \delta'_k$, *such that* $\mathbf{A} = \bigcup_{1 \leq j \leq n} \mathbf{A}_j$. *We say the* n *descriptors are the* n-**interleaved** *descriptors of* $\mathbf{A}$.

Proof : To prove this, we will show that both $\mathbf{A}' \subset \mathbf{A}$ and $\mathbf{A} \subset \mathbf{A}'$. In order to focus on the k-th dimension, without losing any generality, we will fix the values of all indices except the k-th. The set of elements within $\mathbf{A}$ which result have a specific starting point, τ_F, determined by the values we choose for the other indices. First, to prove $\mathbf{A}' \subset \mathbf{A}$, we will show that an arbitrary element of $\mathbf{A}_j$, for any j, is always an element of $\mathbf{A}$. By definition, the m'th element of $\mathbf{A}_j$ has the offset:

$$(m'-1)\delta'_k + \tau_F + (j-1)\delta_k \quad \text{for } 1 \leq m' \leq \frac{\sigma'_k}{\delta'_k} + 1$$

which we can express as $((m'-1)n + j - 1)\delta_k + \tau_F$. This form indicates that the m'th element of $\mathbf{A}_j$ corresponds to the $(m'-1)n + j$th element of $\mathbf{A}$. This is a valid element of $\mathbf{A}$ because, from the assumption, we can show

$$1 \leq (m'-1)n + j \leq \frac{\sigma_k}{\delta_k} + 1$$

since $1 \leq m' \leq \frac{\sigma'_k}{\delta'_k} + 1 = \left\lfloor \frac{\sigma_k - (j-1)\delta_k}{\delta'_k} \right\rfloor + 1$.

Next, to show $\mathbf{A} \subset \mathbf{A}'$, we start from the mth element of $\mathbf{A}$, which has the offset:

$$(m-1)\delta_k + \tau_F \quad \text{for } 1 \leq m \leq \frac{\sigma_k}{\delta_k} + 1.$$

Given two integers x and y such that $x = \left\lfloor \frac{m-1}{n} \right\rfloor$ and $y = (m-1) \ \mathbf{mod}\ n$, we have $m - 1 = nx + y$. Then, the offset of the mth element of $\mathbf{A}$ is transformed as follows:

$$\begin{aligned} (nx+y)\delta_k + \tau_F &= xn\delta_k + \tau_F + y\delta_k \\ &= x\delta'_k + \tau_F + y\delta_k \end{aligned}$$

from which we see that the element corresponds to the $x + 1$th element of descriptor $\mathbf{A}_{y+1}$. We can verify that $\mathbf{A}_{y+1}$ is one of the n descriptors $\mathbf{A}_j$ and that it contains the $x+1$th element since, from the definition of x, y and m, we can show the bounds on x and y are

$$1 \leq x + 1 \leq \frac{\sigma_k}{\delta_k} + 1, \text{ and } 1 \leq y + 1 \leq n.$$

Together, these results show that any element of $\mathbf{A}$ is also an element of $\mathbf{A}'$, and vice versa, which implies $\mathbf{A} = \bigcup_{1 \leq j \leq n} \mathbf{A}_j$. ■

Interleaved accesses are common in unrolled loops, such as those represented with the 3-interleaved descriptors of $\mathcal{X}^2_{u-l+4} + l$ shown in Figure 7. Different processors cause different unrolling depths to be advantageous. Using LMADs, a compiler could automatically identify the interleaved patterns from the loops, as discussed above, and choose an arbitrary n to transform the original loop with n-interleaved accesses. For instance, we can transform the 3-interleaved accesses in Figure 7 to the 4-interleaved accesses for deeper unrolling because we can easily show, for example,

$$\begin{aligned} (\mathcal{X}^2_{u-l+4} + l) \quad &\equiv \quad (\mathcal{X}^8_{u-l} + l) \cup (\mathcal{X}^8_{u-l} + l + 2) \cup \\ &\qquad (\mathcal{X}^8_{u-l} + l + 4) \cup (\mathcal{X}^8_{u-l} + l + 6), \end{aligned}$$

(assuming, for simplicity, that $u - l$ is a multiple of 8) and, from the 4-interleaved LMADs for both arrays x and y, it is straightforward to generate the new code

```
for $i = l$ to $u$ with step 8 do
    $temp = temp + x(i) * y(i) + x(i+2) * y(i+2) +$
            $x(i+4) * y(i+4) + x(i+6) * y(i+6)$
end
```

The property of interleaving also can be used to aggregate multiple contiguous accesses to a single one, such as those represented in Figure 8 by the two LMADs $\mathcal{A}^{5,1}_{15,1}$+2 and $\mathcal{A}^{5}_{15}$+4. The descriptor $\mathcal{A}^{5,1}_{15,1}$+2 consists of its 2-interleaved descriptors $\mathcal{A}^{5,2}_{15,0}$+2 and $\mathcal{A}^{5,2}_{15,0}$+3. Using Theorem 1, $\mathcal{A}^{5}_{15}$+4 can be converted to $\mathcal{A}^{2,5}_{0,15}$+4 to match the dimensions of the other descriptors. Now, we apply Theorem 2 to show that this expanded descriptor and the other two descriptors together form 3-interleaved descriptors of $\mathcal{A}^{5,1}_{15,2}$+2.

In our distributed memory multiprocessor code generation project, mentioned in Section 1.2, the notion of interleaving has been useful to perform three region operations (aggregation, intersection and subtraction) on the LMAD notations, as will be briefly discussed in Section 6, and to determine subregions [24]. For instance, in Figure 8, $\mathcal{A}^{5}_{15} + 4$, which represents a subregion of the region represented by $\mathcal{A}^{1,5}_{3,15} + 3$, is in fact one of the 4-interleaved descriptors of $\mathcal{A}^{1,5}_{3,15} + 3$.

5.3 Contiguous Accesses

Contiguous accesses can be formally defined as follows:

Definition 5 *Given* $\tau \geq \tau'$, *let* $\mathbf{A} = \mathcal{A}^{\delta_1,\cdots,\delta_p,\cdots,\delta_d}_{\sigma_1,\cdots,\sigma_p,\cdots,\sigma_d} + \tau$ *and* $\mathbf{A}' = \mathcal{A}^{\delta'_1,\cdots,\delta'_q,\cdots,\delta'_d}_{\sigma'_1,\cdots,\sigma'_q,\cdots,\sigma'_d} + \tau'$. *The LMADs are* **contiguous**, *denoted by* $\mathbf{A} \bowtie \mathbf{A}'$, *if there exist dimensions* (σ_p, δ_p) *and* (σ'_q, δ'_q), *satisfying the conditions*

1. $\mathbf{A}_p \ // \ \mathbf{A}'_q$,
2. $\delta_p = \delta'_q$,
3. δ_p *divides* $\tau - \tau'$,
4. $\tau - \tau' \leq \sigma'_q + \delta'_q$, *and*
5. *the pth dimension index of* $\mathbf{A}$ *and the qth dimension index of* $\mathbf{A}'$ *do not appear in the expressions for the stride/span pairs of* $\mathbf{A_p}$ *or* $\mathbf{A}'_q$

where $\mathbf{A}_p$ *and* $\mathbf{A}'_q$ *are the p-subLMAD of* $\mathbf{A}$ *and the q-subLMAD of* $\mathbf{A}'$, *respectively. Then,* (σ_p, δ_p) *and* (σ'_q, δ'_q) *are the* **bridge dimension***s for the relation* $\mathbf{A} \bowtie \mathbf{A}'$.

In Figure 8, $\mathcal{A}^{5,1}_{15,1} + 2$ and $\mathcal{A}^{1,5}_{3,15} + 3$ are contiguous by Definition 5. Also, in Figure 3, we can show that the two descriptors $\mathcal{A}^{8,1}_{24,n_b} + 0$ and $\mathcal{A}^{1,8}_{n_b,24} + l$ are contiguous as long as $l \leq n_b + 1$ holds.

Theorem 3 *Let* $\mathbf{A}$ *and* $\mathbf{A}'$ *be the contiguous descriptors defined in Definition 5, and let* $t = \tau - \tau'$*. Then,*

$$\mathbf{A} \cup \mathbf{A}' = \begin{cases} \mathcal{A}^{\cdots,\delta_p,\cdots,\delta_d}_{\cdots,\sigma_p+t,\cdots,\sigma_d} + \tau' & \text{for } \sigma_p + t > \sigma'_q \\ \mathbf{A}' & \text{otherwise.} \end{cases}$$

PROOF: Once the conditions of Definition 5 are met and we have identified dimensions p from $\mathbf{A}$ and q from $\mathbf{A}'$, we know that all strides from $\mathbf{A}$ and $\mathbf{A}'$ are the same, and that $\mathbf{A}$ is shifted to the right from $\mathbf{A}'$. Condition 3 means that $\mathbf{A}$ is shifted an integral number of qth dimension strides from the start of $\mathbf{A}'$. Condition 4 means that the shift leaves no gap between the end of dimension q and the start of $\mathbf{A}$. If $\tau - \tau' < \sigma'_q + \delta'_q$, then the start of dimension p would overlap the end of dimension q, while $\tau - \tau' = \sigma'_q + \delta'_q$ would mean that $\mathbf{A}$ starts immediately at the end of dimension q. If $\sigma_p + t > \sigma'_q$, this serves to extend the span of dimension q. Otherwise, dimension p would be completely inside the span of dimension q and would add nothing to dimension q. By Condition 5, no other dimension uses the pth and qth dimension indices associated with the bridge dimensions, so forming a combined dimension has no implications for the other dimensions. ■

Notice that it follows immediately from Theorem 3 that if two LMADs are found to be contiguous as above, and $\sigma_p + t \leq \sigma'_q$, then $\mathbf{A} \subset \mathbf{A}'$.

Theorem 3 can be used to aggregate contiguous accesses to represent their access patterns with a single LMAD. For instance, $\mathcal{A}^{5,1}_{15,1}+2$ and $\mathcal{A}^{1,5}_{3,15}+3$ in Figure 8 are aggregated to a single descriptor by showing

$$\mathcal{A}^{5,1}_{15,1} + 2 \cup \mathcal{A}^{1,5}_{3,15} + 3 \equiv \mathcal{A}^{1,5}_{4,15} + 2.$$

Similarly, the two descriptors $\mathcal{A}^{8,1}_{24,n_b}+0$ and $\mathcal{A}^{1,8}_{n_b,24}+l$ in Figure 3 can be aggregated to produce $\mathcal{A}^{1,8}_{n_b+l,24}+0$ under the constraint $l \leq n_b+1$.

To determine whether two descriptors are contiguous, we would first sort the dimensions of each by stride, an $O(d \log d)$ process, then in one pass check that at least $d-1$ dimensions match and keep track of which indices are used in which other dimensions. If only one dimension of each differs by the span, we check Conditions 3, 4, and 5. If all dimensions match, then one more pass needs to be made, checking Conditions 3, 4, and 5 for each pair. This process is dominated by the $O(d \log d)$ process.

In Section 5.2, we showed how to use the interleaved structure to aggregate $\mathcal{A}^{5}_{15} + 4$ and $\mathcal{A}^{5,1}_{15,1} + 2$. In fact, we can do the same with Theorem 3. For this, we first must obtain a 2-dimensional descriptor $\mathcal{A}^{\delta^*,5}_{0,15} + 4$, expanded from $\mathcal{A}^{5}_{15} + 4$ using Theorem 1 because the original 1-dimensional LMAD cannot be directly applied to Definition 5. Since δ^* can be any number by definition, we set $\delta^* = 1$ to match the stride of its counterpart pair $(1,1)$. Thereafter, it is again straightforward to show

$$\mathcal{A}^{5,1}_{15,1} + 2 \bowtie \mathcal{A}^{1,5}_{0,15} + 4 \ (\equiv \mathcal{A}^{1,5}_{3,15} + 2).$$

Theorem 4 *Let* $\mathbf{A}$*,* $\mathbf{A}'$ *and* $\mathbf{A}''$ *be LMADs. Assume* $\mathbf{A} \bowtie \mathbf{A}'$ *and* $\mathbf{A}' \bowtie \mathbf{A}''$*, where the bridge dimension in* $\mathbf{A}'$ *for both relations is the same. Then* $(\mathbf{A} \cup \mathbf{A}') \bowtie \mathbf{A}''$ *and* $\mathbf{A} \bowtie (\mathbf{A}' \cup \mathbf{A}'')$*.*

PROOF: Let (δ,σ) in $\mathbf{A}$ and (δ',σ') in $\mathbf{A}'$ be the bridge dimensions for $\mathbf{A} \bowtie \mathbf{A}'$, and (δ',σ') in $\mathbf{A}'$ and (δ'',σ'') in $\mathbf{A}''$ be those for $\mathbf{A}' \bowtie \mathbf{A}''$. Note that, by assumption, the bridge dimensions in $\mathbf{A}'$ for both relations are the same (δ',σ').

We will first prove $\breve{\mathbf{A}} \bowtie \mathbf{A}''$, where we refer to $\mathbf{A}{\cup}\mathbf{A}'$ as $\breve{\mathbf{A}}$. If $\mathbf{A} \subset \mathbf{A}'$, then, trivially, $\mathbf{A}' = \breve{\mathbf{A}}' \bowtie \mathbf{A}''$. If $\mathbf{A} \not\subset \mathbf{A}'$, it is obviously true that Conditions 1, 2, and 5 in Definition 5 must hold between $\breve{\mathbf{A}}$ and $\mathbf{A}''$ because of the fact that, by Theorem 3, $\breve{\mathbf{A}}$ has the same dimensions as $\mathbf{A}'$ except the bridge dimension for $\mathbf{A}{\bowtie}\mathbf{A}'$, and $\delta{=}\delta'{=}\delta''$. For Conditions 3 and 4, let τ, τ', and τ'' be the base offsets of $\mathbf{A}$, $\mathbf{A}'$, and $\mathbf{A}''$, respectively, and let $t_1 = |\tau - \tau'|$ and $t_2 = |\tau' - \tau''|$. We must consider six cases, corresponding to the ordering possibilities of τ, τ', and τ''. In the proof, a $\surd$ denotes conditions that follow trivially from assumptions or simple algebraic manipulation. For instance, in Case 1 below, the $\surd$ next to Condition 3 denotes that it is trivial to prove δ divides $t_1 + t_2$ since t_1 and t_2 are multiples of δ.

$$\begin{array}{llll} \mathbf{1.}\ \tau \leq \tau' \leq \tau'' & \mathbf{A} \bowtie \mathbf{A}' \text{ implies :} & t_1 \leq \sigma + \delta & \\ & \mathbf{A}' \bowtie \mathbf{A}'' \text{ implies :} & t_2 \leq \sigma' + \delta & \\ & \mathbf{A} \cup \mathbf{A}' = \breve{\mathbf{A}}^{\delta}_{\sigma'+t_1} + \tau, & & \\ \text{Thus :} & \delta \text{ divides } t_1 + t_2 & \text{(Cond 3)} & \surd \\ & t_1 + t_2 \leq \sigma' + t_1 + \delta & \text{(Cond 4)} & \surd \end{array}$$

$$\begin{array}{llll} \mathbf{2.}\ \tau \leq \tau'' \leq \tau' & \mathbf{A} \bowtie \mathbf{A}' \text{ implies :} & t_1 \leq \sigma + \delta & \\ & \mathbf{A}' \bowtie \mathbf{A}'' \text{ implies :} & t_2 \leq \sigma'' + \delta & \\ & \mathbf{A} \cup \mathbf{A}' = \breve{\mathbf{A}}^{\delta}_{\sigma'+t_1} + \tau, & & \\ \text{Thus :} & \delta \text{ divides } t_1 - t_2 & \text{(Cond 3)} & \surd \\ & t_1 - t_2 \leq \sigma' + t_1 + \delta & \text{(Cond 4)} & \surd \end{array}$$

$$\begin{array}{llll} \mathbf{3.}\ \tau' \leq \tau'' \leq \tau & \mathbf{A} \bowtie \mathbf{A}' \text{ implies :} & t_1 \leq \sigma' + \delta & \\ & \mathbf{A}' \bowtie \mathbf{A}'' \text{ implies :} & t_2 \leq \sigma' + \delta & \\ & \mathbf{A} \cup \mathbf{A}' = \breve{\mathbf{A}}^{\delta}_{\sigma+t_1} + \tau', & & \\ \text{Thus :} & \delta \text{ divides } t_2 & \text{(Cond 3)} & \surd \\ & t_2 \leq \sigma + t_1 + \delta & \text{(Cond 4)} & \surd \end{array}$$

$$\begin{array}{llll} \mathbf{4.}\ \tau' \leq \tau \leq \tau'' & \mathbf{A} \bowtie \mathbf{A}' \text{ implies :} & t_1 \leq \sigma' + \delta & \\ & \mathbf{A}' \bowtie \mathbf{A}'' \text{ implies :} & t_2 \leq \sigma' + \delta & \\ & \mathbf{A} \ \not\subset \mathbf{A}' \text{ implies :} & \sigma + t_1 > \sigma' & \\ & \mathbf{A} \cup \mathbf{A}' = \breve{\mathbf{A}}^{\delta}_{\sigma+t_1} + \tau', & & \\ \text{Thus :} & \delta \text{ divides } t_2 & \text{(Cond 3)} & \surd \\ & t_2 \leq \sigma + t_1 + \delta & \text{(Cond 4)} & \surd \end{array}$$

$$\begin{array}{llll} \mathbf{5.}\ \tau'' \leq \tau \leq \tau' & \mathbf{A} \bowtie \mathbf{A}' \text{ implies :} & t_1 \leq \sigma + \delta & \\ & \mathbf{A}' \bowtie \mathbf{A}'' \text{ implies :} & t_2 \leq \sigma'' + \delta & \\ & \mathbf{A} \cup \mathbf{A}' = \breve{\mathbf{A}}^{\delta}_{\sigma'+t_1} + \tau, & & \\ \text{Thus :} & \delta \text{ divides } t_2 - t_1 & \text{(Cond 3)} & \surd \\ & t_2 - t_1 \leq \sigma'' + \delta & \text{(Cond 4)} & \surd \end{array}$$

$$\begin{array}{llll} \mathbf{6.}\ \tau'' \leq \tau' \leq \tau & \mathbf{A} \bowtie \mathbf{A}' \text{ implies :} & t_1 \leq \sigma' + \delta & \\ & \mathbf{A}' \bowtie \mathbf{A}'' \text{ implies :} & t_2 \leq \sigma'' + \delta & \\ & \mathbf{A} \cup \mathbf{A}' = \breve{\mathbf{A}}^{\delta}_{\sigma+t_1} + \tau', & & \\ \text{Thus :} & \delta \text{ divides } t_2 & \text{(Cond 3)} & \surd \\ & t_2 \leq \sigma'' + \delta & \text{(Cond 4)} & \surd \end{array}$$

From the above proof, we showed $\mathbf{A} \cup \mathbf{A}'$ and $\mathbf{A}''$ are contiguous. Proving $\mathbf{A} \bowtie \mathbf{A}' \cup \mathbf{A}''$ uses identical logic, so we will omit it. ■

Notice that, given three descriptors $\mathbf{A}$, $\mathbf{A}'$, and $\mathbf{A}''$, $\mathbf{A} \bowtie \mathbf{A}'$ and $\mathbf{A}' \bowtie \mathbf{A}''$ do not necessarily imply $\mathbf{A} \bowtie \mathbf{A}''$. This is because the relation $\bowtie$ is not transitive. To illustrate this, consider Figure 8. $\mathcal{A}^{5}_{15} + 1$ and $\mathcal{A}^{1,5}_{3,15} + 3$

are not contiguous, although $\mathcal{A}^{5}_{15} + 1 \bowtie \mathcal{A}^{5,1}_{15,1} + 2$ and $\mathcal{A}^{5,1}_{15,1} + 2 \bowtie \mathcal{A}^{1,5}_{3,15} + 3$. Therefore, the question is "given a set of pairs of contiguous LMADs, how does one determine the order of aggregation to obtain the optimal result?" To this question, Theorem 4 responds that the result of a computation does not depend on the order in which aggregations are applied as long as they meet the constraint on the bridge dimensions given in Theorem 4. One form of this theorem is illustrated in Figure 11, where three LMADs $\mathbf{A}^{(1)}$, $\mathbf{A}^{(2)}$, and $\mathbf{A}^{(3)}$ and their initial contiguous relations, $\mathbf{A}^{(1)} \bowtie \mathbf{A}^{(2)}$ and $\mathbf{A}^{(2)} \bowtie \mathbf{A}^{(3)}$, are given in a *contiguous relation graph.* Whether $\mathbf{A}^{(1)}$ and $\mathbf{A}^{(2)}$ are aggregated first or $\mathbf{A}^{(2)}$ and $\mathbf{A}^{(3)}$ are aggregated first, we can always obtain the same result $\mathbf{A}^{(123)}$ if the bridge dimensions of $\mathbf{A}^{(2)}$ for both relations are the same. This is the case with the access descriptors in Figure 8, which can be aggregated in any order to produce $\mathcal{A}^{1,5}_{4,15}+1$, which can be further simplified to $\mathcal{A}^{1}_{19}+1$ using coalescing.

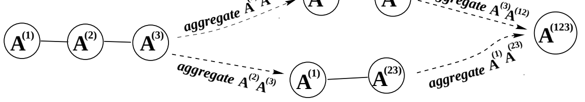

Figure 11: Example of contiguous relation graphs: in the graph, a node $\mathbf{A}^{(i)}$ denotes a LMAD, and an edge the relation $\bowtie$. $\mathbf{A}^{(ij)}$ represents the aggregated region of those represented by $\mathbf{A}^{(i)}$ and $\mathbf{A}^{(j)}$; that is, $\mathbf{A}^{(ij)} \equiv \mathbf{A}^{(i)} \cup \mathbf{A}^{(j)}$.

5.4 Other operations

In addition to the operations described in this paper, we have devised LMAD algorithms for several other region operations [24]: union, intersection, and subtraction. In the cases where these operations cannot be carried out precisely, the resulting descriptor can be marked to be an over- or under-estimation of the actual locations.

5.5 Experimental results showing the degree of simplification

In an attempt to determine how much simplification can be achieved by the techniques of coalescing and contiguous aggregation, we measured the simplification produced by coalescing and contiguous aggregation for our set of test codes. We computed all LMADs interprocedurally without applying any simplification techniques. We counted the total number of LMADs produced at all loop headers and CALL statements, and totaled the number of dimensions used for all LMADs.

Then, we again computed all LMADs interprocedurally, but applied both coalescing and contiguous aggregation iteratively during the process until no more simplification was possible, and recorded the number of LMADS and dimensions used in them.

We chose to show the reduction in the total number of dimensions as a measure to indicate the amount of simplification performed because it captures both the reduction in the number of LMADs (through aggregation) and the reduction of the number of dimensions (through coalescing). The results, presented in Figure 12, show that a significant amount of simplification can be achieved in most cases.

6 Applications of LMADs for Compiler Techniques

Typically, array privatization and dependence analysis [3, 9, 14, 29] are based on array region operations. Thus, their accuracy is heavily dependent on array access analysis. According to our experiments with Polaris, current techniques are limited in some cases by the complexity of subscripts. To illustrate this, consider the loop in Figure 13. To parallelize the I-loop in the example, it is necessary to determine that array Y can be privatized. That is, it must be shown that within each iteration of the outermost loop, every element of array Y is always written before read. The difficulty here is that the subscript expressions for array Y are non-affine, and the accesses are made by multiple indices, J and K. Due to these complications, existing array privatization techniques [21, 31] cannot identify the exact access region for Y; as a consequence, they could not privatize Y and thus would fail to parallelize the loop. We found that the LMAD is often effective to overcome these limitations.

In the loop of Figure 13, for example, the two *write* accesses for array Y are represented with descriptors

$$\mathcal{Y}^{2,2^{\mathtt{J}+1}}_{2^{\mathtt{J}+1}-2,2^{\mathtt{N}+1}-2} + 1 \text{ and } \mathcal{Y}^{2,2^{\mathtt{J}+1}}_{2^{\mathtt{J}+1}-2,2^{\mathtt{N}+1}-2} + 2,$$

from which it is clear that they are *contiguous* (in fact, also *interleaved*). Therefore, they can be *aggregated* into an exact region represented with

$$\mathcal{Y}^{1,2^{\mathtt{J}+1}}_{2^{\mathtt{J}+1}-1,2^{\mathtt{N}+1}-2} + 1,$$

as illustrated in Figure 13, which in turn can be *coalesced* into the equivalent form $\mathcal{Y}^{1}_{2^{\mathtt{N}+2}-3} + 1$.

Despite the complex subscript expressions in the example, using the LMAD representation helps reveal that the write region for Y is as simple as one equivalently represented by Y(1:$2^{\mathtt{N}+2}$ − 2:1) in triplet notation. In our experiments, this simplification allowed existing array privatization techniques to calculate the exact write region of array Y and, similarly, the read region due to two references, Y(K+2**J) and Y(K+2**J+2**(N+1)-1), in the loop. As a result, we can prove that the read region is covered by the write region (in fact, they are the same), and thereby eliminate dependence by declaring Y as *private* to the loop.

The LMAD representation is useful not only for array privatization, but also for other techniques depending on array access analysis, including dependence analysis and the generation of communication primitives such as Send/Receive or Put/Get. For instance, in our code transformation [25] for multiprocessors with physically distributed memory, we used the LMAD to generate Put/Get primitives of the general form

put/get(x(l_x:u_x:s_x),y(l_y:u_y:s_y),p).

The put transfers the elements of x (from $x(l_x)$ to $x(u_x)$ with stride s_x) in local memory to remote address $y(l_y)$ with stride s_y in destination processor p. The get works the same way except that the source and destination of data movement are reversed. Most communication primitives supported in existing languages or machines [8,

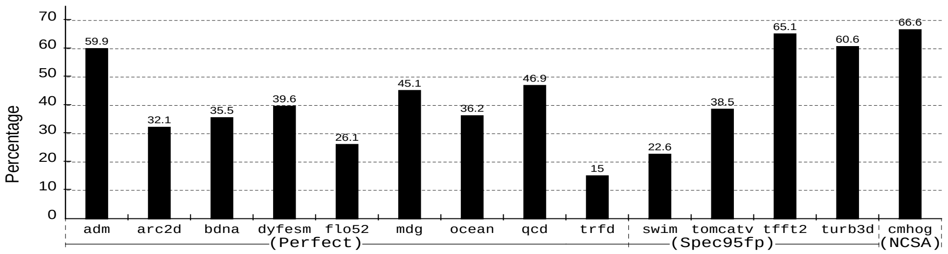

Figure 12: Percentage reduction in total number of LMAD dimensions by coalescing and contiguous aggregation

```
do I = 1, M
  do J = 0, N
    do K = 0, 2**J-1
      Y(2*K+2**(J+1)-1) = ...
      Y(2*K+2**(J+1)) = ...
    enddo
  enddo
  do J = 0, N
    do K = 0, 2**J-1
      ... = Y(K+2**J)
      ... = Y(K+2**J+2**(N+1)-1)
    enddo
  enddo
enddo
```

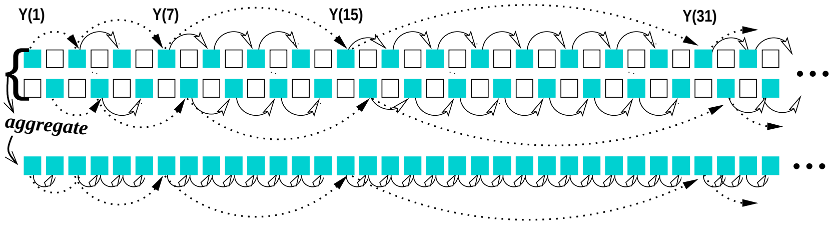

Figure 13: Code example similar to code found in FFT applications, such as `tfft2`

10, 22, 23] require triplet notation for fast vector copying between distributed memories. Without simplification of access patterns, we could not generate efficient Put/Get primitives for codes with complex subscript expressions, such as those shown in Figures 2 and 13. But, by showing that the actual access patterns are just simple consecutive memory accesses, we significantly reduced communication overhead in our target code.

In [24], we presented the experimental evidence that the LMAD can be useful to simplify various access patterns with complex subscripts and, thereby, facilitate the application of compiler techniques. In the experiment, as should be expected, the effectiveness of the LMAD is roughly proportional to the percentage of the complex accesses shown in Figure 4.

processors	2	4	8	16	32	64
`mdg`	1.2	1.3	1.4	1.6	2.6	3.0
`trfd`	1.1	1.0	1.2	1.5	1.5	1.5
`tfft2`	1.6	2.1	3.2	4.7	7.1	7.4

Table 1: Speed increase factor due to the LMAD on the Cray T3D for processors between 2 and 64.

Table 1 shows the results of experiments with three of the programs used to compute the frequencies in Figure 4. The entries in the table represent the ratio of speedup improvement produced by using the LMAD compared to the speedup produced by Polaris using techniques supported by triplet notation. For instance, we were able to improve the original speedup of `trfd` by about 50% on 16 processors (leading to a speed improvement factor of 1.5) when we simplified the array accesses with our techniques before applying Polaris analysis. Not surprisingly, using the LMAD produced no additional speedup on the programs with simple subscripting patterns, such as `swim` and `tomcatv`.

7 The Contribution of This Work

We have extended the work described here so that it works interprocedurally and devised parallelization and array privatization techniques based on the LMAD. The implementation of our techniques is only partially complete, but preliminary tests and hand analysis indicate that the techniques will be able to automatically do interprocedural parallelization and privatization of loop nests within the code `tfft2` from the SPEC95fp benchmarks, despite its usage of non-affine subscript expressions like `X(1+K+I*2**(L-1))`, in which `I`, `K`, and `L` are indices of the surrounding loops. A more detailed description of these techniques and their implementations will be presented in a forthcoming PhD thesis [17].

We believe that our techniques will subsume the existing Polaris intra-procedural parallelization and privatization techniques because of the increased precision of the representation. Furthermore, we believe that we will be able to parallelize interprocedural loop nests because the techniques work across procedure boundaries.

8 Conclusion

A significant portion of the array subscript expressions encountered in the benchmark programs we used are too complex to be precisely representable in triplet notation, yet quite often still lead to simple access patterns. We have shown that a more general array access representation, based on the stride and span produced by each loop index, can accurately represent the accesses.

Furthermore, with the application of polynomial-time algorithms, we can aggregate and simplify the representation of accesses that exhibit common patterns, such as coalesceable, interleaved, and contiguous accesses. The result is often an access pattern that is simple enough to allow the application of efficient region operations used in optimizing compilers for dependence analysis, array privatization, and communication generation. We presented the performance improvements obtained on our target machines by applying these techniques. Results indicate that these techniques hold real promise for optimizing programs.

We believe that the notion of strides and spans, and the simplification techniques based on them, should be useful to other compiler studies that need accurate intra- and inter-procedural array access analysis, regardless of the representation used for their array accesses.

References

[1] V. Balasundaram and K. Kennedy. A Technique for Summarizing Data Access and its Use in Parallelism Enhancing Transformations. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[2] U. Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, Norwell, MA, 1997.

[3] W. Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, June 1995.

[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[5] M. Burke and R. Cytron. Interprocedural Dependence Analysis and Parallelization. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 162–175, July 1986.

[6] D. Callahan and K. Kennedy. Analysis of Interprocedural Side Effects in a Parallel Programming Environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[7] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating Local Address and Communication Sets for Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.

[8] Cray Research Inc. *SHMEM Technical Note for Fortran*, 1994.

[9] B. Creusillet and F. Irigoin. Exact vs. Approximate Array Region Analyses. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1996.

[10] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parallel Programming in Split-C. *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[11] G. Dantzig and B.Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory*, pages 288–297, 1973.

[12] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley Pub. Co., New York, 1989.

[13] J. Grout. Inline Expansion for the Polaris Research Compiler. Master's thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, May 1995.

[14] J. Gu, Z. Li, and G. Lee. Symbolic Array Dataflow Analysis for Array Privatization and Program Parallelization. *Proceedings of Supercomputing '95*, December 1995.

[15] P. Havlak. *Interprocedural Symbolic Analysis.* PhD thesis, Rice University, May 1994.

[16] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, pages 27–45, 1994.

[17] J. Hoeflinger. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, forthcoming.

[18] C. Huson. An In-line Subroutine Expander for Parafrase. Master's thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, May 1982.

[19] Z. Li and P. Yew. Efficient Interprocedural Analysis for Program Parallelization and Restructuring. *Proceedings of the SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, July 1988.

[20] Z. Li, P. Yew, and C. Zhu. An Efficient Data Dependence Analysis for Parallelizing Compilers. *IEEE Transaction on Parallel and Distributed Systems*, 1(1):26–34, January 1990.

[21] D. Maydan, S. Amarasinghe, and M. Lam. Array Data-Flow Analysis and its Use in Array Privatization. *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languges*, January 1993.

[22] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, January 12, 1996.

[23] J. Nielocha, R. Harrison, and R. Littlefield. Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers. *Proceedings of Supercomputing '94*, pages 340–349, November 1994.

[24] Y. Paek. *Automatic Parallelization for Distributed Memory Machines Based on Access Region Analysis.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, April 1997.

[25] Y. Paek and D. Padua. Experimental Study of Compiler Techniques for NUMA Machines. *IEEE International Parallel Processing Symposium & Symposium on Parallel and Distributed Processing*, April 1998.

[26] W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8), August 1992.

[27] W. Pugh and D. Wonnacott. Nonlinear Array Dependence Analysis. Technical Report 123, Univ of Maryland at College Park, November 1994.

[28] Z. Shen, Z. Li, and P. Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. *IEEE Transaction on Parallel and Distributed Systems*, 1(3):350–364, July 1990.

[29] P. Tang. Exact Side Effects for Interprocedural Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[30] R. Triolet, F. Irigoin, and P. Feautrier. Direct Parallelization of Call Statements. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 176–185, 1986.

[31] P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, May 1995.