Access Regions: Toward a Powerful Parallelizing Compiler *

Yunheung Paek, Jay Hoeflinger, David Padua

Department of Computer Science
University of Illinois at Urbana-Champaign,
1304 West Springfield Avenue,
Urbana, IL 61801, USA
{paek,hoefling,padua}@csrd.uiuc.edu

Keywords: Compilers, Aggregation, Parallelization, Privatization, Run-time test, Region Analysis.

Abstract

The bulk of the work within a scientific program involves processing data stored in arrays. We present a general and efficient means of representing the region of an array accessed by a section of a program. We introduce a notation for access regions, and a set of region operations for manipulating them. We show how a region processor which implements our region operations can form the basis for a parallelizer which handles array privatization, run-time parallelization, communication generation, and interprocedural analysis.

1 Introduction

Existing compiler techniques depend heavily on the analysis of array subscripting patterns. Dependence analysis [5] is one example for parallelizing compilers, but *access region analysis* is also crucial for array privatization [4], communication optimization for Non-Uniform Memory Access (NUMA) multiprocessors [2, 3], locality enhancement [8], and interprocedural summarization [9].

Compiler modules implementing such techniques must represent the array accesses in some standard fashion. For instance, Tu and Padua [4] approximated access regions for array privatization with the *triplet notation*. The same notation was used in papers by Tseng [10] and Chatterjee, Gilbert and Long [11] for message generation. Blume and Eigenmann [5] excluded the stride from the triplet notation in their dependence test for simplicity, but at the expense of accuracy. *Convex regions* [14, 17] express the geometrical shape of array accesses. They can be used with Fourier-Motzkin-based dependence tests [21, 22]. Balasundaram and Kennedy [15] simplified the convex region to detect task parallelism.

Such representations are designed to strike a balance between the efficiency of using the representation and its expressiveness. Generally, design decisions leading to these forms have come down on the side of reducing expressiveness, or limiting expressiveness to that needed for a specific compiler module, in order to increase efficiency. But limited expressiveness can prevent compiler transformations. We have

*The research described is supported by Army contract #DABT63-95-C-0097. This work is not necessarily representative of the positions or policies of the Army or the Government.

found that compiler techniques based on traditional access region representations are limited in some important cases, to the point of being unable to carry out the technique.

Based on our experience with the Perfect [28] and SPEC benchmarks, and several full scientific codes [1], we have developed a region access representation which takes advantage of the clear structure inherent in the array accesses of most scientific programs. Our design attempts to allow maximum expressiveness without sacrificing efficiency. We are implementing this representation in **Polaris** [1], which is the parallelizing compiler being developed by the authors and others at Illinois.

2 Motivation

The Polaris parallelizer uses a triplet notation for array access patterns in modules such as the privatizer [4] and the dependence analyzer [6]. Polaris has been successfully obtaining speedups for many scientific applications on a variety of shared-memory multiprocessors, but we have seen that Polaris still fails to obtain good speedups for some applications. We carefully studied these programs, and found one of the reasons is that Polaris was unable to handle several common access patterns occurring in these programs due to the limited expressiveness of the triplet notation, and thus fails to parallelize loops in these programs. For instance, the programs commonly contain complex access patterns such as those involving multiple strides and diagonal access patterns. Also, these access patterns typically depend on many symbolic values in the program, often unknown at compile time. Consider a loop `INTRAF_do1000`, one of the major loops in the MDG benchmark, in Figure 1:

```
do I=1,M,N
   ...
   FX(I)=FX(I)*FHM
   FX(I+2)=FX(I+2)*FHM
   FX(I+1)=FX(I+1)*FOM
   ...
enddo
```

Figure 1: Code example from MDG, after being simplified

Polaris was not able to parallelize this loop due to the unknown value `N`. Even without knowing `N`, it is easy to see that the loop is parallelizable as long as $\texttt{N} > 2$. In order to handle this case, we need a representation which can handle multiple strides, with conditions, and some mechanism to manipulate them to generate predicates for run-time tests. The triplet notation used by Polaris cannot support this manipulation, and so Polaris simply serializes the loop. Without this mechanism, we might have to employ expensive run-time techniques [7, 13], in order to parallelize this loop.

As another example, consider the loop in Figure 2, which can be found in FFT programs such as the TFFT2 benchmark. It is the most important loop in TFFT2 and contains several complications:

- both the `K` and `J` loops are triangular loops,
- the subscripting patterns for `X` is non-affine, and
- the access pattern for `X` has multiple varying strides: 1 and 2^{L-1}.

```
do I = 0, 2**M-1
   do L = 1, (1+M)/2
      do J = 0, 2**(1+M-L)-1
         do K = 1, 2**(L-2)
            ...
            X(K+J*2**(L-1)) = ···
            X(K+J*2**(L-1)+2**(L-2)) = ···
            ...
         enddo
      enddo
      do J = 0, 2**(M-L)-1
         do K = 1, 2**(L-1)
            ...
            ··· = X(K+J*2**(L-1))
            ··· = X(K+J*2**(L-1)+2**M/2)
            ...
         enddo
      enddo
   enddo
enddo
```

Figure 2: Code example from TFFT2, after inlining and induction variable substitution

The portion of the array `X` which is used in this loop is privatizable since the part of the array which is read is completely covered by the part that is written. However, any representation which cannot handle non-affine expressions cannot represent this access region. Although the Polaris dependence analyzer [4, 5] can handle non-affine expressions, it still fails to privatize `X` and parallelize this loop because of the other complexities involved in these access patterns.

In addition, our work on developing compilation techniques [2, 3] for NUMA multiprocessor systems such as the Cray T3D and the Convex Exemplar showed a need for gathering even more precise array access information for supporting efficient data movement and copying between distributed memories. For instance, for data movement in these systems, the communication analyzer often needs to selectively decide between small exact(*MUST*) regions and a large approximate(*MAY*) region in order to reduce remote memory latency. To meet this requirement, an access region representation must support the notion of accuracy.

In our quest for a better access pattern representation, we considered the convex regions, but forms which represent access patterns by sets of constraints typically must use a more general dependence test [22], which cannot handle non-affine expressions [5, 6]. Forms which use the triplet notation cannot handle such complicated access patterns as discussed earlier, but lend themselves to more efficient manipulation in many parts of a compiler. So, we decided to develop a new representation by combining the expressiveness of convex regions with the efficiency of triplet notation, plus information about accuracy.

3 Description of Access Regions

In our approach, we attempt to keep the exact region access pattern as long as possible. Sometimes, we can't avoid losing accuracy and when we do, we mark that the access information is approximate. In the following sections, we will describe this form, define operations on the representation, and discuss

the module being implemented in Polaris to process these representations.

3.1 Components of Access Regions

Within a program section such as a loop or a procedure, a reference to an array X is represented by $X(s(\mathcal{I}))$ where $s(\mathcal{I})$ is the subscript function defined on the set of indices $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ in which each index i_k varies from l_k to u_k with stride s_k, denoted as $[l_k{:}u_k{:}s_k]$, within the program section. $s(\mathcal{I})$ needs not be affine. If X is multi-dimensional, then the subscript expression is *linearized* [24] to generate $s(\mathcal{I})$. The array region $\mathcal{R}$ accessed by the array reference is represented by the four-tuple

$$\mathcal{R} = (\textit{Access Descriptor, Accuracy, Access Type, Predicates})$$

where the access descriptor is represented by two parts, the *access function* and *index ranges*:

$$(f(\mathcal{I}))[i_1 = l_1 : u_1 : s_1][i_2 = l_2 : u_2 : s_2] \cdots [i_m = l_m : u_m : s_m].$$

The accuracy is MUST if $\mathcal{R}$ is accurate. Otherwise, it is MAY. The access type is READ or WRITE, depending on whether $\mathcal{R}$ is read or written by the corresponding reference(s). The predicate is a condition under which $\mathcal{R}$ is valid.

The access function $f(\mathcal{I})$ is the same as $s(\mathcal{I})$, as long as $s(\mathcal{I})$ is a *monotonic function* [6, 25] within the index ranges. Although many subscript functions encountered in scientific programs may not be affine, most are monotonic [5], and those few which are not monotonic may be converted to a monotonic function with a possible accuracy loss. For instance, in Figure 3, the access region for `IV` is ((J-2)[J=1:N:1],MUST,READ,**T**[†]), since `J-2` is clearly monotonic. However, for the access region for `V`, ((IV(J-2))[J=1:N:1],MUST,READ,**T**), we cannot determine whether `IV(J-2)` is monotonic unless we have knowledge about the contents of `IV`. In this case, we may convert the access region for `V` to ((J)[J=VLOW:VHIGH:1],MAY,$\cdots$) with monotonic function `J`, if the accuracy loss is tolerable [3].

Predicates add precision to $\mathcal{R}$. In Figure 3, even when the compiler cannot determine the value of `P`, the access region for `Q` can be represented by ((3I)[I=1:N:1],MUST,WRITE,`P`). Alternatively, for flow-insensitive analysis, we may represent this as ((3I)[I=1:N:1],MAY,WRITE,**T**). Also, when we linearize the subscript expression of the multi-dimensional array `W` in the figure, we might lose the original array dimension information. In order to avoid that, we can extract predicates from the array dimension information to produce the region

$$((2\text{J}+(\text{I-1})\text{M})[\text{J=1:N:1}][\text{I=1:J:1}],\text{MUST,WRITE},1 \leq 2J \leq M).$$

In the operations in Section 3.5, this extra predicate will be used through the *range dictionary* [1, 6] combined with various symbolic manipulation modules to supply accurate symbolic range information.

3.2 Abstract Access Form

We have seen that, for the most part, within limited sections of a program, the access regions of interest have a *regularity* of structure. Furthermore, quite often, related access regions have a *similarity* of

[†] **T** represents TRUE, which means that there is no constraint on this region

```
subroutine foo(X,Y,Z,W,M)
real V(VLOW:VHIGH),W(M,*),X(*),Y(M,*),Z(*) ···
 ···
do J = 1, N
   do I= 1 ,J
      W(2*J,I) = V(IV(J-2))
   enddo
   if (P) then
      Q(3*I) = ···
   endif
enddo
···
do I = 1, N, 1
   do J = 1, N, 1
      do K = 1, 4*J, 2
         X(I+5*J+K) = Z(N*J+I)
      enddo
      Z(J+N*I) = Y(C,D) + Y(I+1,I)
   enddo
enddo
```

Figure 3: Code example

structure. This is true because several references to a single array within a loop nest are generally accessed using the same loop indices and with similar subscript expressions. Experimental evidence for the regularity and similarity of array subscripting may be gleaned from the work of Blume and Eigenmann [5, 6], who note that their dependence test, built to analyze regular access patterns, is essentially as successful as the Omega Test [22], which was built to handle more general patterns.

In attempting to capture the properties of regularity and similarity in our representation, we first define the *span* of the monotonic access function $f(\mathcal{I})$ due to $i_k \in \mathcal{I}$ to be the maximum distance moved by varying only i_k:

$$\delta_{i_k} = |f(i_1, \cdots, i_{k-1}, u_k, i_{k+1}, \cdots, i_m) - f(i_1, \cdots, i_{k-1}, l_k, i_{k+1}, \cdots, i_m)|$$

and the *stride* of the access due to i_k to be

$$\sigma_{i_k} = |f(i_1, \cdots, i_{k-1}, i_k + s_k, i_{k+1}, \cdots, i_m) - f(i_1, \cdots, i_{k-1}, i_k, i_{k+1}, \cdots, i_m)|\,.$$

Using the characteristics of span and stride, the access region $\mathcal{R}$ can be represented by another form, which we call the *abstract access form*, as follows:

$$\mathcal{R} = (\ \mathcal{A}^{\sigma_{i_1},\sigma_{i_2},\cdots,\sigma_{i_m}}_{\delta_{i_1},\delta_{i_2},\cdots,\delta_{i_m}} + l, \mathit{Accuracy,\ Access\ Type,\ Predicates})$$

where l is the lower bound in $\mathcal{R}$. In this form, the access descriptor is represented by a list of the spans and strides where we define $(\sigma_{i_k}, \delta_{i_k})$ to be the k-th *stride/span pair* of $\mathcal{R}$.

Converting the access region to the abstract form helps us to identify the similarity between access patterns. We say that stride/span pairs (σ'_x, δ'_x) and (σ''_y, δ''_y) from two different access regions $\mathcal{R}'$ and $\mathcal{R}''$ *match* if $\sigma'_x = \sigma''_y$ and $\delta'_x = \delta''_y$. We define regions $\mathcal{R}'$ and $\mathcal{R}''$ to be **isomorphic**† if the numbers of

†structurally the same

stride/span pairs in the two regions are the same and each stride/span pair in $\mathcal{R}'$ has a unique matching stride/span pair in $\mathcal{R}''$. For instance, in Figure 3, the abstract access form

$$(\mathcal{A}^{N,1}_{N^2-N,N-1} + N + 1, \text{MUST}, \text{READ}, \mathbf{T})$$

is for the access region of `Z(N*J+I)`, ((NJ+I)[J=1:N:1][I=1:N:1],MUST,READ,**T**). Similarly, the form

$$(\mathcal{A}^{1,N}_{N-1,N^2-N} + N + 1, \text{MUST}, \text{WRITE}, \mathbf{T})$$

is for the access region of `Z(J+N*I)`, ((J+NI)[J=1:N:1][I=1:N:1],MUST,WRITE,**T**). Since we can find corresponding matching stride/span pairs, the two regions are isomorphic. Also, they have the same lower bound $N + 1$, so we can prove that the WRITE region of `Z(J+N*I)` is exactly covered by the READ region of `Z(N*J+I)`.

This abstract access form is not limited by the original array dimensionality. By linearization, for instance, we can represent access patterns for multi-dimensional arrays. In Figure 3, the array `Y` is accessed along a diagonal. We apply linearization to the subscript expression and represent it accurately as $\mathcal{A}^{M+1}_{(N-1)(M+1)}+2$. Traditional triplet notation would express such a diagonal access pattern as `Y(2:N+1:1,1:N:1)`, which is inaccurate. The abstract form here makes it obvious that the access is regular, with a single stride. Also, notice that the access region for the single element `Y(C,D)` can be represented by $\mathcal{A}^{\sigma^*}_0 + \mathtt{C} + (\mathtt{D}-1)\mathtt{M}$ where σ^* can be any integer number.

Sometimes, a span or stride may not be a constant, varying on the values of indices in $\mathcal{I}$. Figure 4 shows the simplified version of code from Figure 2 for clarity.

```
do I = 1, N, 1
   do J = 1, 2**I, 1
      X(2**I+J) = ···
   enddo
enddo
```

Figure 4: Simplified code example from TFFT2 in SPEC 95 benchmarks

The access region for `X(2**I+J)` is $((2^I+\text{J})[\text{I=1:N:1}][\text{J=1:}2^I\text{:1}],\text{MUST},\text{WRITE},\mathbf{T})$, and its corresponding abstract form is $(\mathcal{A}^{2^I,1}_{N,2^I-1}+3,\cdots)$, subject to the index range [I=1:N:1]. Here, σ_I and δ_J vary depending on the value of index `I`, while σ_J and δ_I are a constant or a symbolic constant. This case typically happens in triangular loops or when the subscript functions are non-affine.

We say an index $i \in \mathcal{I}$ is *region-dependent* on index $j \in \mathcal{I}$ if either δ_i or σ_i is an expression containing j. In the example above, `J` is region-dependent on `I` because σ_J contains `I`, and `I` is also region-dependent on itself. As another example in Figure 5, `J` is region-dependent on `I` because δ_J(=I-1) contains `I`.

```
do I = 1, N, 1
   do J = 1, I, 1
      X(I*(I-1)/2+J) = ···
   enddo
enddo
```

Figure 5: Simplified code example from TRFD in Perfect Benchmarks

In order to apply the region operation techniques discussed in Section 3.5, we need to handle the region-dependent indices in $\mathcal{R}$ so that all strides and spans in the access descriptor are constant. In

the next section, we describe how we could handle the region-dependent indices when we generate the abstract access form.

3.3 Generating Abstract Access Form

Although many access regions encountered in scientific applications are made by multiple indices, the aggregated access regions are usually seamless. That is the case where the references to an array make a series of contiguous accesses with one index and jump over those accesses with another index to start a new section of accesses. As an example, let's look at the loop from the SDOT routine in the BLAS library:

```
      do 50 I = M,N,5
         STEMP = STEMP + SX(I)*SY(I) + SX(I+1)*SY(I+1) +
     *      SX(I+2)*SY(I+2) + SX(I+3)*SY(I+3) + SX(I+4)*SY(I+4)
 50   continue
```

The references to both arrays `SX` and `SY` access five contiguous elements with stride 1 and jump the elements with stride 5 to the next section, resulting in seamless access to the arrays from `M`-th element to (`N`+4)-th element. Similar access patterns also can be commonly found in full scientific programs [1]. In these cases of contiguous access, we found that the access pattern made by several indices with various strides can be represented by an access pattern with a single index. For instance, the access region representation ((I+J)[I=1:N:4][J=0:2:2],$\cdots$) has two indices with strides 2 and 4. Here, the index `I` is redundant because we can still represent the same access region only with index `J`, after combining the index ranges: that is, ((J)[J=1:N+2:2],$\cdots$). *Coalescing* is a technique for simplifying the region representation by eliminating these redundant indices as described in Figure 6.

```
coalesce_region(R, i_j, i_k) {
     if (σ_{i_j} divides σ_{i_k} and σ_{i_k} ≤ δ_{i_j} + σ_{i_j}) {
        remove redundant index i_k from R
        if (i_j is region-dependent on i_k)
           δ_{i_j} = δ_{i_j}(u_k) + δ_{i_k}
        else
           δ_{i_j} = δ_{i_j} + δ_{i_k}
     }
  }
```

Figure 6: Algorithm for coalescing region: the form $\delta_{i_j}(u_k)$ refers to the value of the expression δ_{i_j} with u_k substituted for i_k

In the algorithm, the access region $\mathcal{R}$ is of the abstract access form, and indices i_j and i_k are defined in $\mathcal{R}$, with index ranges $[l_j{:}u_j{:}s_j]$ and $[l_k{:}u_k{:}s_k]$, respectively. `coalesce_region` determines whether i_k is redundant. If so, it removes i_k and combines the original index ranges to generate the new range for i_j. In order to determine whether i_k is redundant, the stride of i_k and stride/span pair of i_j must be examined to test if i_j can represent the same access region without i_k, by adjusting the span δ_{i_j}. For example in Figure 3, the access region for `Z(N*J+I)` is $(\mathcal{A}^{N,1}_{N^2-N,N-1} + N + 1,\cdots)$. `coalesce_region` converts the region to a simpler form $(\mathcal{A}^{1}_{N^2-1} + N + 1,\cdots)$, since $\sigma_I(=1)$ divides $\sigma_J(=N)$, and $\sigma_J \leq \delta_I + \sigma_I(=N)$. Similarily, we obtain the same coalesced region for `Z(J+N*I)` in the example.

When we gather array region information from the program and convert the information to an abstract form, we apply coalescing to eliminate unnecessary indices from $\mathcal{R}$. To completely remove all unnecessary indices, we need at most $O(m^2)$ coalescings for a region with m stride/span pairs.

According to our study, coalescing in many cases [6] helps us to remove non-constant strides or spans from the access descriptor in the abstract form. For example, the region $(\mathcal{A}^{2^I,1}_{N,2^I-1}+3,\cdots)$ from TFFT2 in Figure 4 can be coalesced to $(\mathcal{A}^{1}_{2(N+1)-1}+3,\cdots)$ removing the index I, and thus the resulting abstract form contains only constant strides and spans. Similarly, the original region $(\mathcal{A}^{1,I}_{\frac{N^2-N}{2}-1,I-1}+3,\cdots)$ from TRFD in Figure 5 can be coalesced to $(\mathcal{A}^{1}_{\frac{N^2+N}{2}-1}+3,\cdots)$.

In some cases where coalescing could not remove those indices from the access descriptor, we may convert the original access region to a MAY region in order to eliminate them. For instance, if j is region-dependent on a set of indices $i_x,\cdots,i_z$, then the new stride for j is defined to 1, and the new span is defined to be the maximum value of δ_j subject to the index ranges $[i_x{=}l_x{:}u_x{:}s_x]\cdots[i_z{=}l_z{:}u_z{:}s_z]$. Consider the following triangular loop as an example:

```
real X(M,*)
 ...
do I = 1, N, 1
   do J = 1, I, 1
      X(J,I) = ...
   enddo
enddo
```

The access region for `X` ia $(\mathcal{A}^{1,M}_{I-1,(N-1)M}+1,\mathrm{MUST},\cdots)$ after linearization. `coalesce_region` cannot remove the index `I` in the access descriptor, and thus we convert the region to $(\mathcal{A}^{1,M}_{N-1,(N-1)M}+1,\mathrm{MAY},\cdots)$ since $max(I-1)=N-1$ on the range [I=1:N:1]. Although we may lose accuracy of region information here, we have found that, in many flow-insentive analyses including communication analysis, this approximate information can still be useful.

3.4 Region Processor

Many components of a parallelizer rely on the analysis of array access regions. Dependence analysis checks whether the access patterns of arrays overlap. Array privatization, one of the most important dependence-elimination techniques, is based on region *intersection* and *subtraction* operations. Precisely combining regions of access is important for generating the data communication needed for NUMA machines. In this section we describe a *region processor*, which is the module we are implementing in Polaris for supporting the basic Access Region manipulation operations, which are described in Section 3.5.

Since the access regions being operated on will often involve unknown values, the Region Processor is designed to proceed with the operations by making favorable assumptions, and to return the expressions representing those assumptions as conditions under which the result is correct. The condition expressions could be evaluated at runtime, when perfect information is available, to choose between alternative transformations.

The work of the Region Processor is supported by two important features of the Polaris compiler. First, the program is represented in Gated Single Assignment (GSA) form [4]. The GSA form makes it easy to determine which definition of a variable is used at any point in the program, and the conditions

under which a certain definition is used. Second, the symbolic manipulation modules in Polaris, such as range propagation and the range dictionary [1, 6], make the value ranges for variables available at any point in the program. These features provide a mechanism which can determine relationships between variables even when their exact values are unknown. This rich environment was crucial to the success of the Range Test [5], and can enable many of the symbolic operations of the Region Processor.

The predicate for an access region $\mathcal{R}$ may be thought of as the pertinent information found in the gating information from the GSA and in the value range constraints. These conditions and values provide the symbolic manipulation context for making decisions within the region processor. So, implicit in the following operation descriptions is the use of the predicates, which provide the ability to reason symbolically about the relationships between variables.

3.5 Basic operations on Access Regions

Each region operation within the Region Processor is structured as a decision tree as shown in Figure 7. Each decision is a simple one, such as "is the lower bound value of $\mathcal{R}_1$ larger than the upper bound value of $\mathcal{R}_2$". The decision tree structure is also easily extensible by a compiler implementor for improved accuracy by simply adding more branches to the tree.

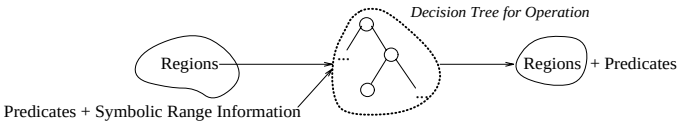

Figure 7: Region Processing

We decided to design the region operations for regions with similar structures. However, if the regions do not meet the similarity constraints, our representation allows us to simplify the regions and still use our operations, or else choose different techniques which might be a better fit [21, 22].

Due to the strict word limit and the intrinsic complexity of the operations, we cannot discuss all the algorithms in this abstract, even though we will include them in the full paper. Therefore, in this section, we focus on one important operation, subtraction, and we will briefly mention the remaining operations.

3.5.1 Subtraction

For the purposes of describing the subtraction algorithm, we will first describe how it works when subtracting single-stride regions, then show how to extend it to multi-stride regions.

Consider the single-stride regions $\mathcal{R}_1$ and $\mathcal{R}_2$ with lower bounds l_1 and l_2, and upper bounds u_1 and u_2, respectively. We define the **distance** between the two regions to be $l_2 - l_1$. By comparing the lower and upper bounds, we can determine the area of $\mathcal{R}_1$ which does not overlap $\mathcal{R}_2$. This area must be part of the result. If there is an area of overlap, then we must calculate the elements which are accessed by both regions, and remove them from $\mathcal{R}_1$. The unremoved elements in $\mathcal{R}_1$ also must be part of the result.

The `subtract_regions` function converts any two single-stride regions to sets of regions with the same stride by finding their **complementary regions** with the appropriate stride. Any single-stride region $\mathcal{R}$ with stride σ can be represented equivalently by n regions, each with stride $n\sigma$. We define

these n regions to be n-complementary regions of $\mathcal{R}$. For example, let $\mathcal{R}$ be represented by the access descriptor $\mathcal{A}^2_{16}+1$. This access pattern can be represented by two complementary regions with forms $\mathcal{A}^4_{16}+1$ and $\mathcal{A}^4_{12}+3$. Similarily, four regions, with forms $\mathcal{A}^8_{16}+1$, $\mathcal{A}^8_8+3$, $\mathcal{A}^8_8+5$, and $\mathcal{A}^8_8+7$, are 4-complementary regions of $\mathcal{R}$.

`subtract_regions` finds a stride σ_{lcm} which is the least common multiple (LCM) of the strides of both $\mathcal{R}_1$ and $\mathcal{R}_2$, then it forms a set of complementary regions with that stride for each original region. Next, it must find which of the complementary regions in those sets access the same elements within the overlapping area. This is done by finding a pair of complementary regions, one from each set, which are separated by a distance equal to a multiple of σ_{lcm}. We say that two such regions are **synchronized**. For instance, $\mathcal{A}^M_{\delta^*}+5$ and $\mathcal{A}^M_{\delta^+}+2M+5$ are synchronized, since $(2M+5)-5$ is a multiple of M, where δ^* and δ^+ are any numbers.

The final result consists of all those complementary regions of $\mathcal{R}_1$ which were not synchronized with any complementary regions of $\mathcal{R}_2$, plus the parts of $\mathcal{R}_1$ from the non-overlapping areas, which were calculated earlier.

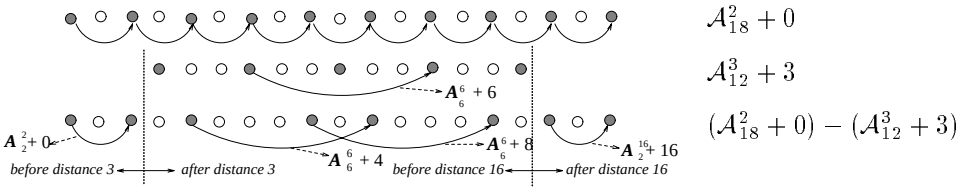

Figure 8: $(\mathcal{A}^2_{18}+0)-(\mathcal{A}^3_{12}+3) \implies \mathcal{A}^2_2+0, \mathcal{A}^6_6+4, \mathcal{A}^6_6+8, \mathcal{A}^2_2+16$

We illustrate the subtraction of two single-stride regions with the example in Figure 8. The task is to subtract $\mathcal{R}_2=\mathcal{A}^3_{12}+3$ from $\mathcal{R}_1=\mathcal{A}^2_{18}+0$. First we find the non-overlapping area in $\mathcal{R}_1$, which is

$$\mathcal{S}_{non-overlap}=\{\mathcal{A}^2_{\lfloor\frac{(3-1)}{2}\rfloor\cdot 2}, \mathcal{A}^2_{\lfloor\frac{(18-(12+3))}{2}\rfloor\cdot 2}+12+3+1\}.$$

Next, we find the $LCM(2,3)$, which is 6. Then, we compute the 3-complementary regions of $\mathcal{R}_1$, which are

$$\mathcal{S}_1=\{\mathcal{A}^6_{\delta^*}+0, \mathcal{A}^6_{\delta^*}+2, \mathcal{A}^6_{\delta^*}+4\}$$

where δ^* is a span not yet computed. Similarly, we find the 2-complementary regions of $\mathcal{R}_2$, which are

$$\mathcal{S}_2=\{\mathcal{A}^6_{\delta^*}+3, \mathcal{A}^6_{\delta^*}+6\}.$$

Comparing $\mathcal{S}_1$ and $\mathcal{S}_2$, and identifying that $\mathcal{A}^6_{\delta^*}+0$ from $\mathcal{R}_1$ and $\mathcal{A}^6_{\delta^*}+6$ from $\mathcal{R}_2$ are synchronized, we remove $\mathcal{A}^6_{\delta^*}+0$ from $\mathcal{S}_1$ to produce

$$\mathcal{S}_{overlap}=\mathcal{S}_1-\{\mathcal{A}^6_{\delta^*}+0\}=\{\mathcal{A}^6_{\delta^*}+2, \mathcal{A}^6_{\delta^*}+4\}.$$

The region $\mathcal{A}^6_{\delta^*}+2$ has a lower bound less than 3, placing it outside the overlapped area, so we recompute its lower bound to place it inside the overlapped area, with the result $\mathcal{A}^6_{\delta^*}+8$. Now, we compute the new spans for the regions in $\mathcal{S}_{overlap}$:

$$\mathcal{A}^6_{\lfloor\frac{12+3-4}{6}\rfloor\cdot 6}+8=\mathcal{A}^6_6+8 \quad \text{and} \quad \mathcal{A}^6_{\lfloor\frac{12+3-8}{6}\rfloor\cdot 6}+4=\mathcal{A}^6_6+4.$$

They are included to make the complete result

$$\mathcal{S}_{result} = \mathcal{S}_{non-overlap} \cup \mathcal{S}_{overlap} = \{\mathcal{A}_6^6 + 8, \mathcal{A}_6^6 + 4, \mathcal{A}_2^2 + 0, \mathcal{A}_2^2 + 16\}.$$

It is very difficult to obtain the exact solution for the general problem of subtraction between arbitrarily-shaped multi-stride access regions. The algorithm to deal with this problem would be too complicated or sometimes even intractable. However, in real cases, the regions of interest are usually regular and have similar shapes, excluding the need for complex algorithms. This leads us to enforce some constraints on the strides and spans of the input regions to simplify the algorithm. For the cases which are too complicated, we can simplify the regions to satisfy the constraints by converting to MAY regions if the application modules (such as those discussed in Section 4) allow a loss of accuracy. If this is not allowed, then we could convert the region to a form which a different technique [22] can handle, since we retain all necessary information in our representation.

Subtraction generalizes in a very natural way to the operation $\mathcal{R}_1 - \mathcal{R}_2$ for multi-stride regions. The non-overlapping parts of $\mathcal{R}_1$ appear in the output just as before, retaining the original shape of $\mathcal{R}_1$. To perform subtraction in the overlapping area for multi-stride access regions, we extend the ideas used to process the single-stride case, such as complementary regions, and synchronized regions. To illustrate the multi-stride case, consider the following loop from the ARC2D benchmark:

```
real X(P,Q,R)
...
do J = 1, N, 1
   do K = 1, M, 1
      X(J,K,2) = ···
      X(J,K,1) = ···
   enddo
   ··· X(J,M,1) ···
   ··· X(J,M,2) ···
enddo
```

Figure 9: Simplified code example from ARC2D in Perfect Benchmarks

Let $\mathcal{R}_{inner}$ be the access region made by the references `X(J,K,1)` and `X(J,K,2)`, and $\mathcal{R}_{outer}$ be the one made by the references `X(J,M,1)` and `X(J,M,2)`. In order to calculate $\mathcal{R}_{inner} - \mathcal{R}_{outer}$, the subscripting expressions are first linearized and aggregated, to produce the abstract regions

$$\mathcal{R}_{inner} = (\mathcal{A}_{N-1,(M-1)P,PQ}^{1,P,PQ} + 1, \cdots) \quad \text{and} \quad \mathcal{R}_{outer} = (\mathcal{A}_{N-1,0,PQ}^{1,P,PQ} + (M-1)P, \cdots).$$

Notice that, in this example, the stride/span pairs are the same between the two regions except for the middle one in each region. By ignoring the other pairs, therefore, we can reduce the multi-stride problem to a single-stride problem, which is $\mathcal{R}'_{inner} - \mathcal{R}'_{outer}$ where

$$\mathcal{R}'_{inner} = (\mathcal{A}_{(M-1)P}^{P} + 1, \cdots) \quad \text{and} \quad \mathcal{R}'_{outer} = (\mathcal{A}_{0}^{P} + (M-1)P, \cdots).$$

The `subtract_regions` function calculates this reduced problem to generate the result

$$\mathcal{S}'_{result} = \mathcal{S}'_{non-overlap} \cup \mathcal{S}'_{overlap} = \{\mathcal{A}_{(M-2)P}^{P} + 1\}.$$

By expanding $\mathcal{S}'_{result}$ for the original multi-stride problem, we have the final result

$$\mathcal{S}_{result} = \{\mathcal{A}_{N-1,(M-2)P,PQ}^{1,P,PQ} + 1\}.$$

3.5.2 Aggregation

Taking a list of regions as input, the `aggregate_regions` function identifies the access patterns of input regions and combines regions with *similar* structure. The first step is to partition the regions into groups based on the similarity of their strides and spans.

The notion of the *similarity* of regions is embodied in various terms, such as *complementary regions* (described in Section 3.5.1), *conjunctive regions* and *subregions*. Two regions are **conjunctive** if their structures are compatible, but shifted by a constant factor, allowing their spans to be combined. An example of conjunctive regions is shown in Figure 10 where the two regions, represented by access descriptors $\mathcal{A}^5_{15} + 1$ and $\mathcal{A}^{1,5}_{2,15} + 2$, are conjunctive because they are shifted by one element and have compatible structure. The aggregated region can be represented by $\mathcal{A}^{1,5}_{3,15} + 1$. We say that $\mathcal{R}_1$ is a **subregion** of $\mathcal{R}_2$ if the elements accessed by $\mathcal{R}_1$ are a proper subset of the elements accessed by $\mathcal{R}_2$. The `aggregate_regions` function determines whether given regions are compatible by comparing stride/span pairs and lower bounds.

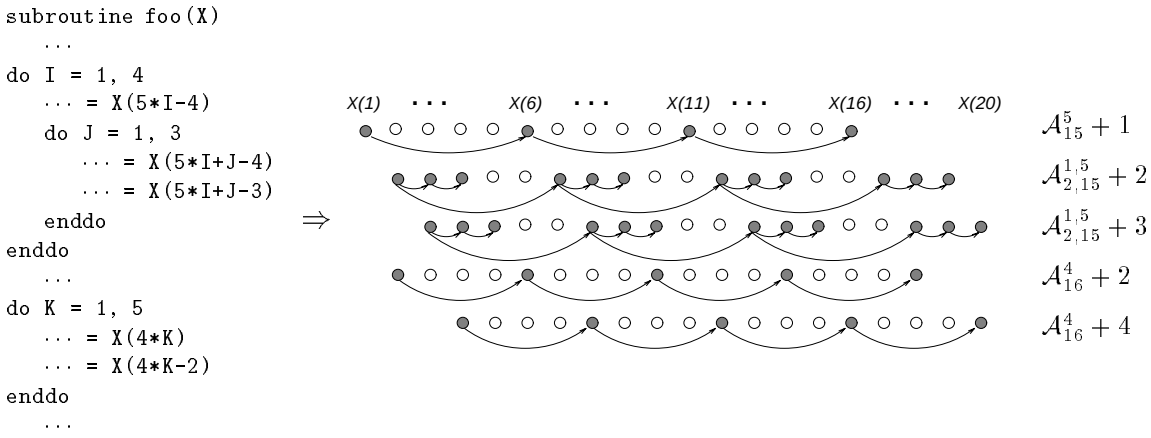

Figure 10: Aggregation of READ accesses for `X` within `foo`

To illustrate the aggregation operation, consider the subroutine `foo` in Figure 10. The access patterns for `X` within `foo` and their corresponding abstract forms are shown in the figure. Given these regions, `aggregate_regions` produces a single aggregated region $\mathcal{A}^1_{19} + 1$ for the summarized MUST READ region for `X` in `foo`, following the procedure:

1. Three conjunctive regions $\mathcal{A}^{1,5}_{0,15} + 1$, $\mathcal{A}^{1,5}_{2,15} + 2$ and $\mathcal{A}^{1,5}_{2,15} + 3$ are aggregated to $\mathcal{A}^{1,5}_{4,15} + 1$.
2. $\mathcal{A}^{1,5}_{4,15} + 1$ is coalesced to $\mathcal{A}^1_{19} + 1$ by the algorithm `coalesce_region`.
3. Two complementary regions $\mathcal{A}^4_{16} + 2$ and $\mathcal{A}^4_{16} + 4$ are aggregated to $\mathcal{A}^2_{18} + 2$ because they are 2-complementary to it.
4. $\mathcal{A}^2_{18} + 2$ and $\mathcal{A}^1_{19} + 1$ are aggregated to $\mathcal{A}^1_{19} + 1$ since $\mathcal{A}^2_{18} + 2$ is the subregion of $\mathcal{A}^1_{19} + 1$.

3.5.3 Intersection

The intersection algorithm for two access regions can return either the exact region(s) of intersection, or a YES/NO answer by checking whether the exact result is empty. The algorithm `intersect_regions` is

a slightly modified version of the `subtract_regions` algorithm. The basic operation is the same, except we produce the synchronized complementary regions as the result.

For example, in Figure 8, suppose we are to perform intersection instead of subtraction on the input regions $((\mathcal{A}^2_{18}+0)\bigcap(\mathcal{A}^3_{12}+3))$. We form the complementary regions for $\mathcal{A}^2_{18}+0$ and $(\mathcal{A}^3_{12}+3)$, as before, then intersect them as before. This gives the result $\{\mathcal{A}^6_6+6\}$.

3.5.4 Uniqueness Test

The uniqueness test determines whether any location is referred to more than once due to the access pattern of the region. The basic mechanism of this test is to check whether all inner spans are within the next-outermost stride. In order to do this, it must be possible to symbolically sort the stride/span pairs according to the stride. Inner spans are those whose strides are smaller. As an example of the `uniqueness` algorithm, consider the abstract form $\{\mathcal{A}^{1,5}_{2,15}+2\}$ displayed as part of Figure 10. Since the inner span is less than the outer stride, we say that the region has the uniqueness property. It is clear from the figure that no location is repeated in that access region.

4 Applications of Access Regions

The array privatizer, dependence analyzer, and communications generation modules can all make use of the Region Processor. The advantages of this are many. First, it should greatly simplify each of these modules, removing all region-handling code from them, letting them concentrate on strategies for using the results of the region processing and the conditions produced by it. Second, it promotes a demand-driven style of compilation, as opposed to a pass-based style. The use of a consistent region representation and framework makes it possible to pass Access Regions between several compiler modules, which are called as they are needed. Third, the conditions generated by the Region Processor make it possible to parallelize loops at run-time instead of serializing them for lack of information.

Each compiler module uses the `aggregate_regions` function to combine the array accesses of interest in a program section. The program section could be a loop-nest, a subroutine or any part of the input program. The module then can use the Region Processor to perform any of the other operations on the aggregated regions. If the Region Processor lacks the information it needs to perform the operation, it returns a condition under which the result is correct.

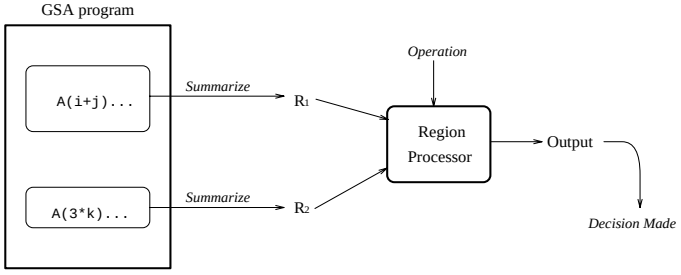

Figure 11: Use of Region Processor

This framework could also be used as a basis for interprocedural analysis. The regions of each array

which are accessed in a subroutine could be summarized in the form of an Access Region, then used in the analysis just like any other Access Region.

4.1 Dependence Analysis

The Access Region representation allows us to solve the problems which we presented in Section 2. For the loop from TFFT2, shown in Figure 2, we can employ `coalesce_region` to summarize the regions accessed at each loop level. The region of `X` written in the first `J` loop may be calculated by first aggregating the writes, then coalescing the regions, to produce the abstract form:

$$(\mathcal{A}^{1}_{2^M} + 1, \text{MUST}, \text{WRITE}, \mathbf{T}).$$

Similarly, the region of `X` read in the second `J` loop becomes:

$$(\mathcal{A}^{1}_{2^M} + 1, \text{MUST}, \text{READ}, \mathbf{T}).$$

So, by subtracting the WRITE region from the READ region, we get an empty region, showing that the array `X` is privatizable in the `I` loop.

For the example in Figure 1, we need the condition extraction function of the Region Processor. The Region Processor can easily extract the predicate needed in the course of using the `uniqueness` function to test for output dependences in the loop. In the example, the aggregated write region for `FX` in the loop is

$$(\mathcal{A}^{1,N}_{2,2(M-1)} + 1, \text{MUST}, \text{WRITE}, \mathbf{T}).$$

The uniqueness test checks whether the inner span `2` is less than the outer stride `N`. Since it can't know that in this case, the Region Processor would report that it is unique under the predicate "$\texttt{N} > 2$". The region test can then produce code which will check the predicate at run time, choosing between a parallel version of the loop and a serial version.

4.2 Communication Analysis

Single-sided communication protocols [18, 19, 23, 27] in the form of PUT/GET primitives have been rapidly gaining wide acceptance. A great advantage of PUT/GET primitives is that their use of asynchronous data communication works well with the shared-memory programming paradigm, which is also assumed by Polaris.

PUT/GETs are useful for removing anti and output dependences as illustrated in [16]. By using `coalesce_region` and `aggregate_regions`, in the loop shown in Figure 12, the Region Processor calculates the region of upwards exposed uses [20] of array `V` as

$$\mathcal{A}^{1}_{MN+N-((t-1)N+1)} + (t-1)N + 1$$

for each iteration $\texttt{I} = t$. The write region for the same iteration is

$$\mathcal{A}^{1}_{N-1} + (t-1)N + 1.$$

The write region for all the following iterations from $\texttt{I} = t+1$ to `M` is

$$\mathcal{A}^{1}_{MN+N-(tN+1)} + tN + 1.$$

The `intersect_regions` function would report an overlap between $\mathcal{A}^1_{MN+N-((t-1)N+1)} + (t-1)N + 1$ and $\mathcal{A}^1_{MN+N-(tN+1)} + tN + 1$, implying an anti dependence. To make the loop nest parallel, we can eliminate the dependence by privatizing the array `V` and generating a GET for the upwards exposed use region.

```
do I=1,M
   do K=1,N
      do L=I,1+M-I
         ... = V(K+(I-1)*N+L*N)
      enddo
      V(K+(I-1)*N) = V(K+(I-1)*N) ...
   enddo
enddo
```

Figure 12: Code example from MDG, with the induction variables substituted

We also use PUT/GETs to implement the *data copying scheme* [2, 3] in SPMD parallel codes for NUMA multiprocessors. In the scheme, we use shared memory as a repository of values for use in private memory. Before a parallel loop starts, the processors copy all data that is used in the loop from shared memory into private memory. After the loop execution completes, the processors copy the results back to shared memory so that all the processors have access to the results. By doing so, we can localize most of the data that are used by the processors in the computations.

In the data copying scheme, gathering precise array access information into a flexible representation is essential for supporting efficient copy(PUT/GET) operations. Our recent experiments with benchmarks showed that our implementation of the scheme, based on our new representation, has been successful, as shown in Figure 13.

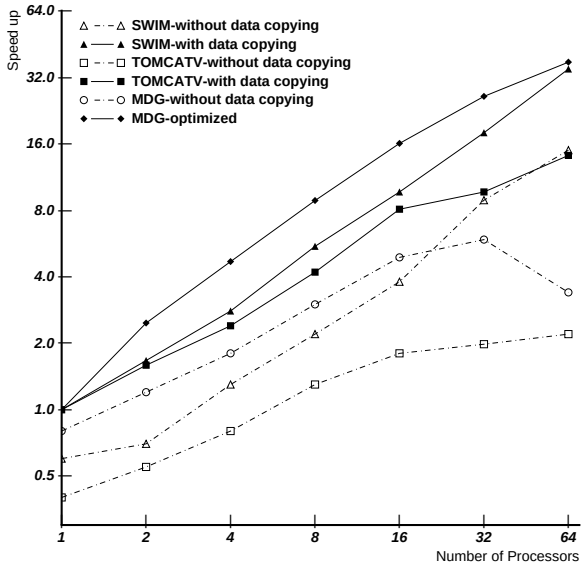

Figure 13: Performance comparisons of automatic parallelization with and without the data copying scheme on the Cray T3D

5 Conclusion

We have presented a new way to represent the access patterns of arrays within a program which is more precise than traditional triplet notation. We have defined operations for manipulating this representation which are suitable for supporting the region processing needs of several compiler modules.

We discussed using a general facility for processing this representation as a basis for privatization analysis, dependence analysis, communication generation, and interprocedural analysis within a parallelizing compiler. We discussed techniques which allow such a processor to return conditions under which the result of the region operation is correct.

References

[1] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, P. Tu, Advanced Program Restructuring for High-Performance Computers with Polaris, *To appear in IEEE Computer*, Dec. 1996, OR. *Technical Report 1473*, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., Jan. 1996

[2] Y. Paek, D. Padua, Automatic Parallelization for Non-cache Coherent Multiprocessors, *Proceedings of 9th Workshop on Language and Compilers for Parallel Computing, OR. To appear in Lecture Notes in Computer Science*, Spring-Verlag, NY, Aug. 1996

[3] Y. Paek, D. Padua, Compiling for Scalable Multiprocessors with Polaris, *To appear in Parallel Processing Letters*, World Scientific Publishing, UK, 1997

[4] P. Tu, D. Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. *Proceedings of the 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995

[5] W. Blume, R. Eigenmann, The Range Test: A Dependence Test for Symbolic Non-linear Expression, *SuperComputing '94 Proceedings*, Nov. 1994, pp. 643-656

[6] W. Blume, Symbolic Analysis techniques for Effective Automatic Parallelization, PhD Thesis, University of Illinois at Urbana-Champaign, June 1995

[7] L. Rauchwerger, D. Padua, The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization, *Proceedings of ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, Jun. 1995

[8] F. Bodin, E. Granston, T. Montaut, Loop Transformations to Prevent False Sharing, International Journal of Parallel Programming.

[9] D. Callahan, K. Kennedy, Analysis of Interprocedural Side Effects in a Parallel Programming Environment

[10] C. Tseng, An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines, PhD Thesis, Rice University, Jan. 1993

[11] S. Chatterjee, J. Gilbert, F. Long, Generating Local Address and Communication Sets for Data-Parallel Programs, *Proceedings of ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Diego, May 1993, pp. 149-158

[12] V. Balasundaram, K. Kennedy, A Techniques for Summarizing Data Access and its Use in Parallelism Enhancing Transformations, *Proceedings of ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, Jun. 1989

[13] J. Saitz, R. Mirchandaney, K Crowley, Run-time Parallelization and Scheduling of Loops, IEEE Trans. Computers, Vol 40, May 1991

[14] R. Triolet, F. Irigoin, P. Feautrier, Direct Parallelization of CALL Statements, *Proceedings of ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, July 1986, pp. 176-185

[15] V. Balasundaram, K. Kennedy, A Technique for Summerizing Data Access and its Use in Parallelism-Enhancing Transformation, *Proceedings of ACM SIGPLAN '89 Conf. on Programming Language and Design and Implementation*, Portland, OR, June 1989

[16] R. Cytron, J. Ferrante, What's in a Name?, *Proceedings of the 1987 International Conference on Parallel Processing*, Aug. 1987, pp. 19-27

[17] B. Creusillet, F. Irigoin, Exact vs. Approximate Array Region Analyses, *Proceedings of 9th Workshop on Language and Compilers for Parallel Computing*, Aug. 1996.

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

[19] J. Nielocha, R. Harrison, R. Littlefield, Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers, *Supercomputing '94 Proceedings*, 1994, pp.340-349

[20] A. Aho, R. Sethi, J. Ullman, Compilers Principles, Techniques, and Tools, Addison-Wesley, CA, 1986

[21] P. Tang, Exact Side Effects for Interprocedural Dependence Analysis, Communications of the ACM, Vol. 35, No. 8, Aug. 1992, pp. 102-114.

[22] W. Pugh, A Practical Algorithm for Exact Array Dependence Analysis, *Communications of the ACM*, Vol. 35, No. 8, Aug. 1992

[23] D. Culler, et al., Parallel Programming in Split-C,*Supercomputing '93 Proceedings*, 1993

[24] M. Burke, R. Cytron, Interprocedural Dedependence Analysis and Parallelization, *Proceedings of ACM SGIPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, July 1986, pp 162-175

[25] H. Zima, B. Chapman, Supercompilers for Parallel and Vector Computers, ACM Press, 1992

[26] CRAY T3D System Architecture Overview, Cray Research, 1993

[27] SHMEM Technical Note for Fortran, Cray Research, Oct. 1994

[28] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, J. Martin, The Perfect Club Benchmarks: Effective Performance Evalution of Supercomputers, *Int'l. Journal of Supercomputer Applications*, Fall 1989, Vol. 3, No. 3, Fall 1989, pp. 5-40