Precise Analysis of Array Usage in Scientific Programs*

M. MANJUNATHAIAH AND DENIS A. NICOLE

Department of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ; e-mail: {mm93r,dan}@ecs.soton.ac.uk

ABSTRACT

The automatic transformation of sequential programs for efficient execution on parallel computers involves a number of analyses and restructurings of the input. Some of these analyses are based on computing array sections, a compact description of a range of array elements. Array sections describe the set of array elements that are either read or written by program statements. These sections can be compactly represented using shape descriptors such as regular sections, simple sections, or generalized convex regions. However, binary operations such as *Union* performed on these representations do not satisfy a straightforward closure property, e.g., if the operands to Union are convex, the result may be nonconvex. Approximations are resorted to in order to satisfy this closure property. These approximations introduce imprecision in the analyses and, furthermore, the imprecisions resulting from successive operations have a cumulative effect. Delayed merging is a technique suggested and used in some of the existing analyses to minimize the effects of approximation. However, this technique does not guarantee an exact solution in a general setting. This article presents a generalized technique to precisely compute Union which can overcome these imprecisions.

1 INTRODUCTION

The automatic transformation of sequential programs for efficient execution on parallel computers involves a number of analyses and restructurings of the input. Traditionally, compiler analysis for such automatic syntheses of parallel programs has been confined to the discovery of parallelism in loops. However, recent research studies demonstrate that loop-level parallelization alone is not adequate to extract good performance from current scalable parallel machines [1, 2]. Compiler analysis is therefore being extended beyond loop boundaries to procedures, including the whole program. Global analysis techniques such as interprocedural dependence analysis [2–4] and array data flow analysis [5, 6] have been introduced to discover coarse grain parallelism, asynchronous computations that perform a significant amount of work between synchronization events. A number of such global analyses techniques are based on computing array sections, a compact description of a range of array elements. Array sections describe the set of array elements that are either read from or written to by program statements.

Array section analysis involves computing side effects. A read-side-effect (RSE) occurs in a subcomputation if an object (such as an array) defined outside

Received September 1995
Revised March 1996

* Supported by a research grant from the Commonwealth Scholarship Commission in the U.K.

Scientific Programming, Vol. 6, pp. 229–242 (1997)
CCC 1058-9244/97/020229-14

the scope of the subcomputation is accessed during the subcomputation.* Similarly, a write-side-effect (WSE) occurs in a subcomputation if an object defined outside the scope of a subcomputation is modified during the subcomputation.

1.1 Representing Array Access Sets

Several techniques have been proposed for representing array sections. They fundamentally differ in the granularity of recording references; how multiple references within a program's region (usually the entire procedure) to the same array are described.

Fine Grain Representations

Methods which generate accurate information on side effects are based on fine grain descriptions. They record each reference to an array separately without attempting to summarize. Descriptors are stored as lists of references; translation (translation refers to transfer of array access information at call sites from the context of the called to caller) and intersection are performed on an element-by-element basis. Two such methods are Linearization [8] and Atom Images [9]. These accurate methods are expensive because they maintain complete information about a procedure's array access sets. While translation has $O(n)$ time complexity, intersection has $O(n^2)$ time complexity for n recorded references. Reference lists, although precise in representing access information, are asymptotically as expensive as the in-line expansion technique.

Coarse Grain Representations

To circumvent the efficiency problems associated with accurate information generation, array sections summary techniques have been proposed which in practice produce good results. The main idea in these techniques is to summarize all the references within a region using suitable coarse grain descriptions. A number of such summary techniques have been proposed in the literature which are based on representing array accesses in terms of convex regions. Union and intersection operations on regions are defined to summarize multiple references and to test for data overlaps, respectively. Three convex region representations proposed are (a) regular sections [4, 10], (b) simple sections [3], and (c) generalized convex regions [11].

As a consequence of summarizing information, the descriptor size becomes independent of the number of references occurring within a region. Translation and intersection each has a time complexity which is a function of the rank of the array, typically linear or quadratic. The generalized convex region representation is an exception where testing for feasibility of intersection is known to be expensive [12]. Moreover, it has been found that the access shapes in a majority of scientific and engineering applications can be precisely represented using such shape descriptors [4, 13]. These properties make the region methods efficient and attractive for practical systems. Although the summary techniques are efficient, they suffer from certain sources of inaccuracies. Summarizing multiple references is performed by applying the binary operation Union on section descriptors. This operation does not satisfy the closure property and therefore approximate solutions are computed so as to remain in the convex representation framework. In addition, the imprecisions resulting from successive operations have a cumulative effect. These approximations can potentially lower the precision of the analysis. The only alternative suggested and used in existing analyses to reduce these approximations is delayed merging [14, 15]. This scheme is based on maintaining a list of descriptors as opposed to a single descriptor. The length of the list has to be a preset limit in the analyses. While a particular limit might work well for some inputs, it might be inadequate for others. If this limit is reached, then merging is enforced. Hence, approximations may persist. Therefore, this scheme does not guarantee an exact solution in general.

This article presents a generalized technique to represent precisely the union of two simple sections. The rest of the article is organized as follows. Section 2 discusses the approximations that are resorted to in existing analyses and presents an alternate method of representation that can avoid these approximations. The framework required for deriving this representation is presented in Section 3. The algorithm for performing Union under this representation and its complexity is dealt with in Section 4. The intersection operation in this representation is presented in Section 5. An example is given in Section 6 to demonstrate the utility of this representation. A discussion on the effectiveness of this new representation is presented in Section 7.

2 COMPLEMENTARY ARRAY SECTIONS

In order to accumulate information about array sections, combination of section descriptors such as Union are frequently performed. The operator Union for merging two section descriptors is a set operation and

* According to the terminology defined in [7].

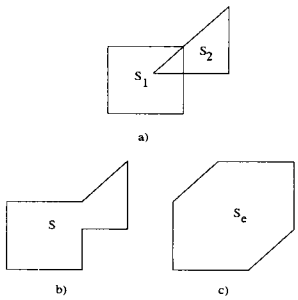

FIGURE 1 Union of sections.

when applied to such section descriptors has the following property:

> *Union:* The binary operator Union (∪) performs a merge of two convex sections. The merge may result in a nonconvex description and therefore the set is not closed under this operator.

2.1 Convex Approximations

As an example to demonstrate the closure property of the operator Union, consider the convex polytopes shown in Figure 1a.† Their exact union is shown in Figure 1b. The basic property of the descriptors used for representing such array sections is that they collectively define a convex polytope. Whenever an operation results in a nonconvex description, the smallest convex approximation is computed in order to remain in the simple section representation framework. Using this approximation criterion, the approximation to the exact union is computed as shown in Figure 1c.

2.2 Accurate Representation Using Complementary Sections

In order to obtain a precise description of array sections under the binary operator Union, a different method of representing array sections, termed complementary sections, is proposed. The fundamental principle governing complementary sections is based on the following observation:

> **Observation 2.1** Any nonconvex region can always be described using a finite number of convex descriptors.

This observation is explained further with an example. Consider again the polytopes shown in Figure 1. The precise union is a nonconvex region (Figure 1b). Therefore an approximate union (S^e) (Figure 1c) is computed to remain in the simple section representation framework. This approximate region can be decomposed into two parts:

1. The precise union (which is nonconvex)
2. A residual set of convex regions

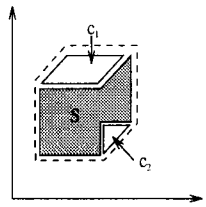

FIGURE 2 Complementary sections.

Figure 2 shows such a decomposition. S represents the precise union (which is non-convex) and C_1, C_2 are the convex regions belonging to the residual part of the approximate union. The outer dotted line is the boundary of the approximate union. The convex combination Union can be expressed in terms of these decompositons as outlined in the following subsection, "Operator Union."

Operator Union

Define the Union of two convex regions as follows:

$$S = S_1 + S_2$$
$$= S^e - S^c$$

In this equation, S^e is the envelope of two convex regions S_1 and S_2. S^c is a set of regions that are not a part of the exact union, but are contained in the envelope. In Figure 2, S^c consists of two regions C_1, C_2. The "difference" of the two regions S^e, S^c will result in the exact union S. Assuming that S^c can be computed and also that each element of S^c is a convex descriptor, we have a description of the precise union S in terms of a number of convex descriptions given by S^e and S^c. The notion that S^e and S^c are in some sense complementary sets which can generate the exact union S is the reason for calling this representation complementary sections. The Union of two sections in this representation is precise.

† The polytopes are represented using simple sections.

3 PROBLEM FORMULATION

The basic idea in the complementary sections framework is to avoid information loss from approximate convex combination operations. This is achieved by extracting the "excess" regions (complements) from the overapproximations and maintaining them as a set of convex regions.

The solution to the precise union operator requires the following two main subproblems to be solved:

1. *Complement construction:* To compute the points that do not belong to the exact union (complements) but appear in the approximated union.
2. *Complement decomposition:* To represent the complements using convex descriptions.

The subproblems are discussed under the simple section representation which is briefly described below. However, in theory, the complementary section framework can be extended to generalized convex representations with increased complexity. From a practical viewpoint, it is interesting to solve the simple section case. Since regular sections are a subset of simple sections, the complementary section framework will be applicable to regular sections as well.

3.1 Simple Section Representation

The simple section representation was developed by Balasundaram and Kennedy [3]. A brief description of the simple section representation is presented here to aid the discussion. A detailed description of related concepts, algorithms, and proofs regarding simple sections can be found in [14].

A simple section is an n-dimensional convex polytope with boundaries of the following types:

$$x_i = c, x_i + x_j = c, x_i - x_j = c$$

where $1 \leq i, j \leq n$, $i \neq j$ and c is a constant.

These conditions imply that a boundary is either parallel to a coordinate axis or at 45° to a pair of coordinate axes. Such a boundary is termed a simple boundary. Thus, a simple section is characterized by simple boundaries. Such a representation enables the precise handling of interesting array sections with convex polyhedral shapes such as rectangles, triangles, banded diagonals, and trapezoids. For example, the references to array A in the subroutine `Access`

```
      SUBROUTINE AccessShapes
      DOUBLE PRECISION  A(100,100)
      DO 10 I = 1, 10
        DO 10 J = 1, 10
          A(I,J) = 1
10    CONTINUE
      DO 20 I = 1, 10
        DO 20 J = 1, I
          A(I+4,J+4) = 1
20    CONTINUE
      RETURN
      END
```

FIGURE 3 Simple sections.

`Shapes` shown in Figure 3 have the simple section representation shown in Figure 1a. The reference in the first nest of `DO` loops corresponds to region S_1 and in the second nest to region S_2.

In the following sections, the solution for the two-dimensional complementary sections is presented. The reason is that the algorithms used in the two subproblems namely complement construction and complement decomposition are applicable to two-dimensional geometry. This assumption clarifies the use of terminology such as polygons and trapezoidalization in the following discussions. The symbols $\star$, $+$, $-$ denote the precise binary operators Intersection, Union, and Difference, respectively, whereas $\cup$ and $\cap$ denote approximate binary operations for Union and Intersection, respectively.

3.2 Complement Construction

The envelope (convex hull) of two sections is computed using the Union algorithm designed for simple section. This results in another simple section which may be the exact union or an approximation. In the latter case, the points that do not belong to the actual union are "extracted" from the envelope through a series of "difference" operations. A procedure to perform these operations is outlined in Figure 4.

l_1 and l_2 are lists with each element containing a descriptor for a polygon, because each step (1, 2) in the above procedure can potentially output a list of polygons. The final output from this procedure is a list of polygons. These polygons may not be convex and further decomposition may be required.

The set operations such as difference of two polygons employed in the complement construction procedure are based on the region finding algorithm of Nievergelt and Preparata [16] (an implementation of these set operations is available in the XYZ GeoBench soft-

```
Procedure ComplementConstruction
Input:  Simple Sections S1 and S2 and
their union Se
Output:  Sc, the set of complements.

   begin
1    ℓ1 ← Se − S1
2    ℓ2 ← ℓ1 − S2
3    Sc ← ℓ2
   end
```

FIGURE 4 `Complement Construction`.

ware [17]). For the example simple sections and its union shown in Figure 1c, the regions identified by this algorithm would be $R_0, R_1, R_2, R_3, R_4, R_5$ as shown in Figure 5. The outer dotted boundary is the region R_0. The two difference operations in procedure `ComplementConstruction` will output R_3 and R_5 which are the desired complementary regions.

All regions computed by the region finding algorithm are represented by edges derived from the input polygons. This leads to an important property of complements as stated below:

Statement 3.1 If the edges of the input polygons are k-oriented‡ with $k = \{0, 1, -1, \infty\}$, the edges in the complements also belong to this class.

3.3 Complement Decomposition

The `ComplementConstruction` procedure outputs a list of polygons, some of which may be convex and others nonconvex. These two types of output need different treatment. In particular, the nonconvex complements need further decomposition.

3.4 Convex Complements

For those complements which are convex, there is no further processing required as demonstrated by the following lemma:

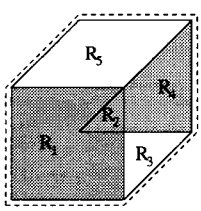

FIGURE 5 Polygonal regions.

‡ Orientation in this context means slope of a line.

Lemma 3.1 If a complement is convex then it is a simple section.

Proof: A simple section is defined as a convex polytope with simple boundaries. From Statement 3.1 the edges in the complement are guaranteed to be simple boundaries. If this complement is convex, then it is a simple section.

The implication of this lemma is that we can test each complement to check if it satisfies the convexity criterion. Checking for convexity is an $O(n)$ time complexity operation for n vertices and therefore can be used as a preprocessing step in the complement decomposition procedure (discussed below). This optimization can reduce the overall time required to construct complements.

3.5 Nonconvex Complements

A complement is not always a convex polytope. In order to remain in the convex representation framework, we need to decompose a composite nonconvex complement into a finite number of convex polytopes. The motivation for performing this decomposition is two fold:

1. Computations on general polygons are difficult but easy for certain primitive shapes such as simple sections. Therefore, it is advantageous to decompose into primitive shapes, perform computations on these well-defined shapes, and combine the results.
2. We would like to remain in the simple section framework to apply the existing convex combination algorithms for these primitive shapes.

Based on the above considerations, we are interested in a decomposition§ with the following properties:

1. Edges in the decompositions should be closed under the orientations of the edges of the chosen representation. Since our chosen representation is simple section, the orientations of the edges in the decompositions should belong to {0, 1, −1, ∞}.
2. The number of partitions should be minimized.

The first property is a stricter requirement than the second. It ensures that the partitions that we obtain can be represented as simple sections. The second

§ Partition is synonymous.

property only affects the time complexity of testing the intersection of two complementary sections.

Under the constraints for partitions mentioned above, the decomposition problem can be stated as follows:

Problem 3.1 Find an optimal convex partition‖ of a simple polygon such that the edges of both the original (input) and the partitioned polygons are k-oriented with $k = 4$; the orientations being $\{0, 1, -1, \infty\}$.

3.6 Decomposition Method

The problem of decomposing a polygon into convex subpolygons has recieved much attention in the field of computational geometry and various algorithms exist depending on the nature of solution. There are at least three methods for decomposing a nonconvex polygon into a finite number of convex polygons: (a) triangulation, (b) trapezoidalization, and (c) optimal convex partitioning. (Triangulation and trapezoidalization can be viewed as special cases of a partition into convex polygons.) If n denotes, the number of vertices of a polygon, then triangulation can be done in $O(n)$ time [18], trapezoidalization is $O(nlogn)$ [19], and an optimal convex partitioning is $O(n^3)$ [20]. The following lemma states the decomposition method that can satisfy the desired closure property for edges in the partition:

Lemma 3.2 A trapezoidal decomposition of a complement exists which results in partitions that are simple sections.

Proof: The complement construction procedure only adds line segments with orientations $\{0, 1, -1, \infty\}$ (from Statement 3.1). A horizontal (or vertical) trapezoidalization of this polygon partitions it into convex quadrilaterals (trapezoids) by adding horizontal (or vertical) line segments. This will only introduce line segments whose orientations are 0 (or ∞ respectively). Since all line segments for each of these partitioned polygon are closed under the orientations and all the partitioned polygons are convex, the decomposition method results in partitions that are simple sections (from Lemma 3.1).

3.7 Decomposition Algorithm

There are several alternative approaches to trapezoidal decomposition with the same asymptotic time complexity. The main ideas of horizontal trapezoidalization are described here. This is an abridged version of the description that appears in [19].

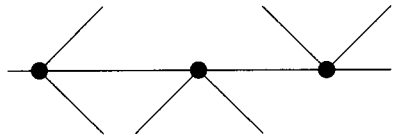

FIGURE 6 Type of vertices.

Given a simple polygon as input, this algorithm generates a set of disjoint trapezoids which covers the polygon. A nonoverlapping decomposition of a polygon is called partitioning (if overlapping is allowed, it is called a covering). In a horizontal partitioning, the parallel edges of the trapezoids are parallel to the x-axis, in the usual intuitive sense of an x-y coordinate system. The nonparallel (vertical) sides of the trapezoids are derived from the edges of the original polygon. The basic idea is to identify these bounding edges of a trapezoid. There are two important characteristics upon which the decomposition strategy, and consequently the identification of the bounding edges, is based:

1. The vertices of the polygon are the defining points for the parallel edges of the trapezoids.
2. The vertices of a simple polygon can be characterized into three types with respect to a horizontal line passing through a vertex. Figure 6 shows these three types.

An ordered scan of the vertices of the polygon is performed. At each vertex encountered during this scan, a horizontal line is passed through it. This line defines a parallel edge (characteristic A) of a trapezoid. Whether this edge defines a starting edge (the edge which initiates one or more trapezoids) or the ending edge (which completes the initiated trapezoids) of a trapezoid can be decided based on the type of vertex (characteristic B). Every time an ending edge is established, a pair of vertical sides have to be found to complete the trapezoid. These vertical sides are those edges of the polygon which are in the left and right neighborhood of the vertex that defined the ending edge. These edges are efficiently computed using dynamic data structures such as height-balanced trees. If we assume that the vertices of the polygon have unique y-coordinates, then the scan starts and termi-

‖ Minimum in the number of partitions.

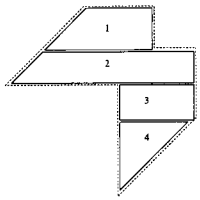

FIGURE 7 Trapezoidal partitions.

nates at unique points. Figure 7 shows a simple polygon and its trapezoidal partitions. The running time of this algorithm is $O(nlogn)$ mainly contributed by the vertex sorting step.

How many partitions are produced by this decomposition algorithm? A theorem due to Asano et al. [21] demonstrates that this kind of partitioning is optimal; it produces the minimum number of partitions for a given class of polygons and the number of trapezoids is given by the index, $t(P)$, of the polygon P.

$$t(P) = n(P) + w(P) - h(P) - 1.$$

Here $n(P)$, $w(P)$, and $h(P)$ represent the number of vertices, windows or holes (a window is a polygon enclosed within an outer polygon), and horizontal edges of P, respectively. Since we are dealing with simple polygons without windows, $w(P) = 0$. For the polygon shown in Figure 7, $n(P) = 8$, $w(P) = 0$, and $h(P) = 3$. Hence, $t(P) = 4$.

3.8 Formal Definition of Complementary Sections

An informal definition of complementary sections was given in Section 2.2. Here we formalize the definition of complementary sections.

Definition 3.1 A complementary section $S' = (S^e, S)$ is a pair consisting of a bounding convex region S^e, the envelope, and a set of disjoint convex regions $S = \{S^c\}$, the complements. S^e and elements of S are represented using simple section descriptors.

4 THE PRECISE UNION ALGORITHM

With the basic machinery for constructing complements and decomposing them into convex descriptions in place, we can now construct the algorithm to compute the union of two complementary sections.

4.1 Union

Given two complementary sections S'_1 and S'_2 as input, the algorithm CompUnion computes the union $S'_1 + S'_2$ which is another complementary section $S'_{new} = (S^e_{new}, S_{new})$ such that $S^e_{new} = S^e_1 \cup S^e_2$ and S_{new} constitute the new set of complementary regions.

```
Algorithm CompUnion
Input: two complementary sections S'1
and S'2.
Output: S'new, the merged complementary
section.

    begin
1    S^e_new ← Envelope (S^e_1, S^e_2)
2    S_polys ← ConsComplements (S^e_new, S'1, S'2)
3    S_new ← Map (Decompose, S_polys)
    end

Procedure ConsComplements (S^e, S'1, S'2)
Input: S^e the envelope, and two
complementary sections S'1 and S'2
Output: the updated set of complements
as a list of polygons S_polys.

   begin
1   ℓ1 ← S^e − S^e_1
2   ℓ2 ← S^e − S^e_2
3   case (ℓ1, ℓ2) of
     /* S^e_1 = S^e_2 = S^e */
4    a) : ℓ1 = nil ∧ ℓ2 = nil
5      S_polys ← UpdateOverlaps (S1, S2)
     /* case S'2 ⊂ S'1 and S^e = S^e_1 */
6    b) : ℓ1 = nil ∧ ℓ2 <> nil
7      S't1 ← UpdateComplements (S1, S^e_2)
8      S't2 ← UpdateOverlaps (S1, S2)
9      S_polys ← SetUnion (S't1, S't2)
     /* case S1 ⊂ S2 and S^e = S^e_2 */
8    c) : ℓ2 = nil ∧ ℓ1 <> nil
9      S't1 ← UpdateComplements (S2, S^e_1)
10     S't2 ← UpdateOverlaps (S1, S2)
11     S_polys ← SetUnion (S't1, S't2)
     /* one is not subset of other */
12   d) : ℓ1 <> nil ∧ ℓ2 <> nil
13     ℓ2 ← ℓ1 − S^e_2
14     S't1 ← UpdateComplements (S1, S^e_2)
15     S't2 ← UpdateComplements (S2, S^e_1)
16     S't3 ← UpdateOverlaps (S1, S2)
17     S_polys ← SetUnion (S't1, S't2 S't3, l2)
18  end case
19  return S_polys
   end
```

```
Procedure UpdateComplements (S, S^e)
Output: S_new, the updated set of
complements.

   begin
1   for each element S^c in S
2    if (S^c ⊄ S^e)
       /* accummulate into S_new */
3      S_new ← SetUnion (S_new, S^c − S^e)
4   return S_new
   end

Procedure UpdateOverlaps (S_1, S_2)
Output: S_∩ the updated set of
complements.

   begin
1   for each element S_1^c in S_1
2    for each element S_2^c in S_2
     /* accummulate if S_1^c ∩ S_2^c ≠ ∅ */
3   S_∩ ← SetUnion (S_∩, S_1^c ∩ S_2^c)
4   return S_∩
   end

Procedure Decompose (P)
Input: A simple polygon P
Output: TrapezList, the partitions of
P into trapezoids

   begin
1   if (IsConvex (P) == true) then
2      TrapezList ← P
3   else
4     TrapezList ← Trapezoidalize (P)
5   return TrapezList
   end
```

Algorithm CompUnion has three components

1. Envelope computes the convex hull of two simple sections using the existing Union algorithm for simple sections.
2. ConsComplements computes the points that do not belong to the exact union but appear in the approximated union. It returns a list of polygons.
3. Map is a higher order function which takes a function Decompose and a list as arguments and applies that function to each element in the list. The output after map operation is a list of simple sections (trapezoids). This higher-order function has been used only for brevity in syntax.

Procedure ConsComplements includes condition checks to track different kinds of overlaps between two sections whose union is being computed. But the principal method to compute complements remains the same as shown in Figure 4. Two kinds of overlaps can be identified which form the basis of the condition checks: (a) Containment—when one section is contained in the other and (b) no containment, but overlap could exist. When one section is contained in the other, their union is the larger of the two. This containment shows up when the difference operations are performed. As shown in the procedure, if only the first difference operation returns a null list of complements then we can declare that $S^e = S_1^c$ and therefore $S_2' \subset S_1'$. In this containment situation, the complements of S_1' have to be updated. Similarly, if the second difference operation results in a null list, then the complements of S_2' are updated. In the case where one is not contained in the other, both complement sets have to be updated. All these updates are performed in procedures UpdateComplements and UpdateOverlaps which involve a series of difference and intersection operations as a two-step process:

1. In the procedure UpdateComplements, the overlap between the complements (S) and the envelope of the section (S^e) is removed by applying difference operations.
2. The overlap between S and the complements of S' cannot be detected in the previous step. This is computed in procedure UpdateOverlaps.

The procedure Decompose checks if the input is convex in which case it does not perform further decomposition. This check is an optimization whose legality was established in Lemma 3.1. Only nonconvex polygons are further processed by procedure Trapezoidalize.

4.2 Complexity

The running time of algorithm CompUnion is evaluated here. The running time is mainly dependent on the number of complements that are to be processed. This is an input-sensitive feature because the number of complements depends on the type and distribution of array sections in the input program. Hence, the running times are estimated assuming some average size, say k, of the complement set. However, the algorithms Union, Intersection, Difference, and Trapezoidalize have fixed cost and the estimates can be expressed in terms of these fixed costs.

Let D = $2d^2$ (where d is the dimension of the array), $|S_1| = k_1$, and $|S_2| = k_2$. For a two-dimensional array

($d = 2$), the cost for Union ($\cup$) or Intersection ($\cap$) is D [14]. The costs for Trapezoidalize and Difference depend on the number of vertices to be processed. The number of vertices to be processed is of the order D and therefore the cost of Trapezoidalize is $O(DlogD)$ and the cost of Difference in UpdateComplements is $O(DlogD)$ (because operands are always convex sections).

Considering only the major cost components, the approximate cost for `CompUnion` is given by the following equation

$$\begin{aligned} \text{C(Comp Union)} \approx\ & (k_1 + k_2)\,\text{D}\log\text{D} \\ & + 2(k_1 k_2)\,\text{D} \\ & + (k_1 + k_2)\,\text{D}\log\text{D} \end{aligned}$$

The terms in the cost expression above are contributed by procedures `ConsComplement` and `Decompose`. Maximum time is involved when lines 14 to 16 of `ConsComplement` are executed. The first term in the cost expression is contributed by `UpdateComplements` in lines 14 and 15. The second term of the cost expression is contributed by `UpdateOverlaps` in line 16. The `ConsComplement` procedure returns a complement set of size proportional to $k_1 + k_2$ which may require further decomposition. Hence, `Map` will require the time shown as the last term of the cost expression.

5 THE PRECISE INTERSECTION ALGORITHM

The basic goal of this article has been to demonstrate a method to compute exactly the Union of convex sections. However, in many analyses the binary operation Intersection is also required. Here, we discuss the intersection algorithm in the complementary sections framework.

5.1 Intersection

Given two complementary sections S_1' and S_2' as input, the algorithm `CompIntersection` computes the "true" intersection $S_1' \star S_2'$. The result is a boolean value indicating the presence or absence of an overlap between S_1' and S_2'.

```
Algorithm CompIntersection
Input: two complementary sections S1'
and S2'
Output: true if sections intersect false
otherwise

    begin
1    S∩ = Intersection(S1', S2)
2    If (S∩ = ∅) return false
3    If (S1 = ∅ ∧ S2 = ∅) return true
4    If (S1 = ∅ ∧ S2 ≠ ∅)
5       return(ScanComplements(S∩, S2))
6    If (S2 = ∅ ∧ S1 ≠ ∅)
        return(ScanComplements(S∩, S2))
7    /* scan both complements */
8    If (S1 ≠ ∅ ∧ S2 ≠ ∅)
9       b1 = ScanComplements(S∩, S1)
10      if (b1 = false) return false
11      else
12        b2 = ScanComplements(S∩, S2)
13      if (b2 = ∅) return false
14      else return true
    end

Procedure ScanComplements (S∩, S)
Input: A simple section S∩ and S the
set of complements
Output: true if intersection exist false
otherwise

    begin
1    for each complement S^c ∈ S
3      S∩ = S∩ − S^c
4      if (S∩ == ∅) return false
5    end for
5    return true;
    end
```

The main steps in `CompIntersection` proceed in the following sequence. The first step computes the intersection of the envelopes using the Intersection algorithm designed for simple sections. If the intersection returns *null* then there is no overlap between the two sections. Otherwise, it indicates the possible existence of an overlap. However, this could be a "false" overlap. To confirm this, the list of complements has to be scanned. This search constitutes the remaining steps of the algorithm which invoke the scan procedure.

The first two steps attempt to establish a null intersection of two sections using the envelopes. Under such conditions, the intersection test will incur the same cost as testing the intersection of two simple sections. This is the advantage of retaining the envelopes in the complementary sections framework. Further steps of the algorithm are required only when a non-null intersection is reported in the first step. In these additional steps, `CompIntersection` essentially has to scan the list of complements to disprove overlap between two sections. This is based on the idea that there is a

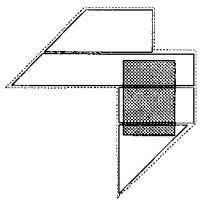

FIGURE 8 A false Intersection.

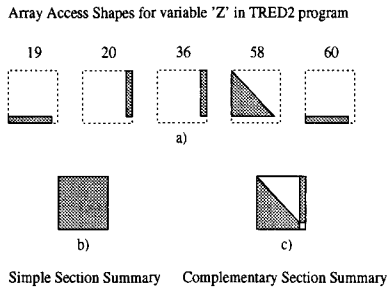

FIGURE 9 Write accesses to Z.

false overlap when one section is contained in the complementary region of the other section. Otherwise, it is a "true" overlap.

The procedure `ScanComplements` basically checks whether a given section is contained in a complementary region. Since the complementary region is maintained as a finite number of convex sections and a given section can potentially span multiple complements, the scan has to be iterated through each complement in turn. At each step in the scan, the part of the section which overlaps with the current complement being tested is removed by a difference operation. If the given section is contained in the complementary region, then a series of such difference operations will eventually terminate in a null region. Figure 8 shows such a false overlap scenario in which S' is the set of complements (trapezoids) and $S_\cap$ is the rectangular region which overlaps this complementary region. As can be noted from Figure 8 the overlap spans multiple complementary sections.

The cost for `CompIntersection` can be computed in a similar manner as that of the Union. Maximum time is involved when lines 9–13 are executed. Each difference operation in procedure `ScanComplements` has time complexity $O((n + s)logn)$, where n is the total number of vertices in the two polygons and s denotes the number of intersections of the line segments of the polygons [16]. After each difference operation, the resulting polygon can have additional vertices. Since n is $O(D)$ for k difference operations, the number of additional vertices is proportional to kD. Hence, the cost for `ScanComplements`, C_{diff} is $O((kD + s)logkD)$. Based on this, the total cost for executing lines 9–13 in `CompIntersection`, considering only the major cost components, is given by the following equation

$$\mathrm{C(CompIntersection)} \approx (k_1 + k_2)\, C_{diff}$$

6 EXAMPLE

We consider an example to demonstrate the use of array section analysis based on complementary sections. This example of computing array access summaries arises in the optimization phase of an automatic parallelizing compiler. Such summary information could be useful for array data-flow analysis.

The write accesses to array variable Z occurring within one iteration of an outer loop are shown in Figure 9a. This loop corresponds to one of the main loops of the TRED2 subroutine from the Eispack library package. The numbers on top of the figures indicate the line numbers corresponding to each update (these numbers are based on a code listing which appears in [22]).

The access shapes corresponding to these updates can be precisely represented using simple sections. However, if these accesses are summarized, then the resulting summary gets approximated as shown in Figure 9b with the upper triangle information lost in the summary. This information can, however, be retained as a complement in the complementary section framework as shown in Figure 9c. If we compare the descriptor sizes required to describe the write accesses to Z, then accurate information of these accesses based on atom images would require five descriptors. Allowing summarization under the simple section framework requires only one descriptor, but at the cost of precision. However, with three descriptors (two complements and one envelope) under complementation, an accurage summary is obtained. More importantly, one can observe from the figure that the complementary sections can describe a nonconvex union precisely using a set of convex descriptors.

7 DISCUSSION

A number of analyses used in an automatic parallelizing compiler are based on computing array sections which are represented using convex shape descriptors. The inaccuracies arising from convex combinations when these descriptors are used in an analysis have been reported in many research studies. In a recent

publication by Creusillet and Irigoin [6], the authors suggest that alternative representation may be required to get better precision when convex descriptors are used. To our knowledge, this is the first attempt at constructing a representation for compile-time analysis of arrays that can produce exact solutions to the closure property of the convex combination operator Union.

7.1 Comments on Effectiveness of Complementary Sections

The often-quoted demerit of the convex regions approach is that the existing techniques produce approximations to the Union of two sections which may weaken the precision of the analysis. Therefore, alternate approaches such as reference lists have been advocated [23]. The approximations can be controlled to a certain extent using delayed merging technique, but this is not a general technique. In this article, we have demonstrated that convex combinations such as Union can be performed accurately and a general technique has been proposed. It only remains to be shown that our approach is efficient. An empirical study has to be conducted to demonstrate the efficiency factor. Here, we make a few general comments on how complementary sections can be effective:

1. The base representation chosen in the complementary section framework for representing array sections is simple sections. Of the three convex region methods proposed in literature, the simple section representation offers a good balance between precision and efficiency. Its Union computation has a worst case complexity which is quadratic in the rank of the array.
2. Empirical studies have demonstrated that nonzero coefficients of loop indexes in most subscripts are either 1 or -1 [13]. Hence, simple section representation is precise enough to represent most array access sets.
3. The computation of complements is based on two standard algorithms from the field of computational geometry, namely region finding and trapezoidalization, both with an asymptotic effort $O(nlog\ n)$.
4. Because many applications found in practice have a maximum of four or five dimensions [13], these fixed cost algorithms can be computed in constant time.
5. Typically, while accumulating array access information using a reference list approach such as Atom Images, the size of the reference list keeps increasing. However, the complement set may not have the same behavior and may tend to diminish if sections begin to overlap with complements. This behavior hinges on a "space filling" hypothesis which is based on the observation that different regions of a program access different sections of an array which together make up the entire array. Hence, as the accumulation of sections occurring across procedure boundaries proceeds, and the sections overlap with the complements, the number of complements will reduce.
6. Because the envelopes are retained along with the complements, the test for intersection will incur the same cost as for testing simple sections in cases where the envelopes are nonoverlapping.

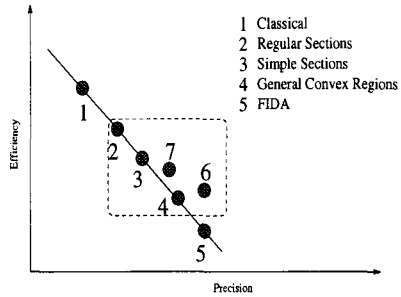

FIGURE 10 Efficiency vs. precision graph.

7.2 Precision Spectrum

Figure 10 shows a representative graph of efficiency vs. precision of different array section analyses. The classical method treats arrays as monolithic units. Hence, it does not distinguish between access to a single element of an array and access to the entire array. Therefore, it lies in the lower end of the precision spectrum. However, the side effect computation based on such summary is highly efficient [24]. At the higher end of the precision spectrum are methods such as FIDA (which combines Atom Images and Linearization). This precision is achieved at the cost of efficiency [23]. Between these extremes in the precision spectrum lie the convex region methods (shown in the dotted enclosure). These methods attempt to strike a balance between precision and efficiency. The complementary section representation adopts the region method and aims at enhancing its precision.

The precision of complementary sections is bounded by points 3 and 5 in Figure 10. This is because the precision can be no worse than simple sections and can be as good as FIDA. There are two

scenarios for positioning the complementary sections in Figure 10:

1. As evidenced by empirical studies (comment 2 on effectiveness in previous section), in the majority of cases arising in practice, the base representation, namely simple section, is adequate to precisely record an array access set (point 4 will coincide with 3 in such cases). However, point 5 is better placed in the precision dimension because it does not suffer from convex combination problems. The complementary sections also overcome the convex combination problem and therefore occupies position 6. The efficiency at this point is depicted to be better than FIDA. This is possible if the pruning property of the complement set discussed in the previous section (comment 5 on effectiveness) holds. This is where we believe the complementary sections lie based on the comments on effectiveness (specifically 2, 4, and 5) in the previous section.
2. In cases where the precision of the base representation is inadequate, the position of complementary section will get shifted to point 7. Although its precision is lowered, it will still be better than the base representation in eliminating other sources of imprecision. Based on indications from empirical studies (comment 2 in previous section), this inadequacy problem will not arise frequently in practice.

The proposition that complementary sections can be efficient in practice can be verified in an actual implementation in the following way. A complementary section consists of one summary information (Envelope) and a set of complements both represented as simple sections. Suppose that the length of this list of complements is k. If we contrast with the reference list approach, the complementary sections will perform better if the cardinality of reference list is greater than k.

8 FUTURE WORK

A prototype implementation is underway in order to assess the effectiveness of the proposed technique. This implementation is being integrated into the SUIF parallelizing compiler [25]. SUIF generates an abstract syntax tree as the intermediate representation of an input upon which our analysis module is built. The approach adopted in this implementation exercise is as follows. We compute the array access shapes and their unions for each nested loop in the body of a procedure. We can then use this information as input to the XYZ GeoBench Software system [17] to compute the complements.¶

The base representation, namely simple section, can be smoothly integrated into a compilation system. Our prior integration effort has been to develop a prototype implementation of this base representation in the SPOC compilation system [26, 27]. The purpose of this implementation was to develop an efficient array usage analysis module to overcome certain deficiencies in existing compiler analysis for the Occam programming language. The array usage analysis module is required to track anamolous parallel updates to shared variables. This implementation adds about 2000 lines of specification code into the compilation system. The specifications are processed by the GMD Compiler Construction Toolset [28] to produce an executable unit. The application of complementary sections to enhance the precision in this analysis is also being pursued.

9 CONCLUSION

A number of important analyses in a parallelizing compiler are based on computing array sections which are represented as convex regions. Although these region methods achieve efficiency, they suffer from certain inaccuracies. A particular source of inaccuracy arises when two sections are merged. We have presented a technique which overcomes this inaccuracy. While alternatives such as delayed merging might work well in certain cases, it does not generalize and approximations could persist. However, the complementary sections framework is an exact solution in a general setting. Its efficiency, however, needs to be assessed through empirical studies.

The discovery of available parallelism in a program is a fundamental precondition for automatic parallelization to be effective. Any mapping transformations can only be as effective aas the precision obtained in this parallelism discovery process. Hence, precise analysis of a program is vital for automatically synthesizing parallel program from its sequential specification. The technique described in this article aims to improve the precision. There are a number of analyses which are based on representing array access sets using convex descriptors such as interprocedural depen-

¶ The XYZ GeoBench Software is written in Object Pascal which inhibits direct integration into our implementation. Integration requires translation into C++ which is cumbersome and a matter of detail.

dence analysis [4, 6], array data flow analysis [29], automatic data partitioning [30], communication analysis [31], analysis for locality optimizations [32], program transformations for reducing false sharing on shared memory multiprocessors [15], and use of array sections information in run-time environments [33]. If these analyses are being impaired due to the imprecision of convex operations, then the precision of all these analyses can be potentially enhanced by adopting the complementary section framework.

ACKNOWLEDGMENTS

Thanks to Mark Keil and Bill Jones of the University of Saskatchewan for their input on the decomposition issues. Thanks to Michelle of ETH Zurich for answering queries on the XYZ GeoBench software. Finally, many thanks to Mrs. Rajini Sivaram for her avid discussions on many issues in this article.

REFERENCES

[1] W. Blume and R. Eigenmann, "Performance analysis of parallelizing compilers on the perfect benchmarks programs," *IEEE Trans. Parallel Distrib. Systems*, vol. 3, pp. 643–656, Nov. 1992.

[2] M. W. Hall, et al., "Overview of an interprocedural automatic parallelization system," in *Fifth Workshop on Compilers for Parallel Computers*, 1995.

[3] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1989, p. 41.

[4] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Trans. Parallel Distrib. Systems*, vol. 2, pp. 350–360, July 1991.

[5] H. Tsalapatas, "Interprocedural array side effect analysis," Master's Thesis, Rice University, Houston, 1994.

[6] B. Creusillet and F. Irigoin, "Interprocedural array region analyses," in *Eighth Workshop on Languages and Compilers for Parallel Computing*, 1995.

[7] W. L. Harrison, "The interprocedural analysis and automatic parallelization of scheme programs," *Lisp Symbolic Computation*, vol. 2, pp. 179–396, 1989.

[8] M. Burke and R. Cytron, "Interprocdural dependence analysis and parallelization," in *Proc. SIGPLAN Symp. Compiler Construction*, 1986, p. 162.

[9] Z. Li and P.-C. Yew, "Efficient interprocedural analysis for program parallelization and restructuring," in *SIGPLAN*, 1988, p. 85.

[10] D. Callahan and K. Kennedy, "Analysis of interprocedural side effects in a parallel programming environment," *J. Parallel Distrib. Comput.* vol. 5, pp. 517–550, 1988.

[11] R. Triolet, et al., "Direct parallelization of CALL statements," in *Proc. SIGPLAN Symp. Compiler Construction*, 1986, p. 176.

[12] F. Irigoin, et al., "Semantic interprocedural parallelization: An overview of the PIPS project," in *Proc. Int. Conf. Supercomputing*, 1991.

[13] Z. Shen, et al., "An empirical study of Fortran programs for parallelizing compilers," *IEEE Trans. Parallel Distrib. Systems*, pp. 356–364, July 1990.

[14] V. Balasundaram, "Interactive parallelization of numerical scientific programs, PhD Thesis, Rise University, Houston, April 1989.

[15] T. E. Jeremiassen and S. J. Eggers, "Reducing false sharing on shared memory multiprocessors through compile time data transformations," in *Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 1995, p. 179.

[16] J. Nievergelt and F. P. Preparata, "Plane sweeping algorithms for intersecting geometric figures," *CACM*, vol. 25, pp. 739–747, Oct. 1982.

[17] Informatik, ETH, Zurich, XYZ GeoBench Manual V4.4.6, Aug. 1995.

[18] B. Chazelle, "Triangulating a simple polygon in linear time," *Disc. Computational Geometry*, vol. 6, pp. 485–524, 1991.

[19] A. Fournier and D. Y. Montuno, "Triangulating simple polygons and equivalent problems," *ACM Trans. Graphics*, vol. 3, pp. 153–174, 1984.

[20] B. Chazelle and D. P. Dobkin, *Computational Geometry*. North Holland: Elsevier Science Publishers, 1985.

[21] T. Asano, T. Asano, and H. Imai, "Partitioning a polygonal region into trapezoids," *J. Assoc. Computing Machinery*, vol. 33, pp. 290–312, 1986.

[22] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel Distrib. Systems*, vol. 3, pp. 179–193, March 1992.

[23] M. Hind, et al., "An empirical study of precise interprocedural array analysis," *J. Sci. Prog.* vol. 3, pp. 255–271, 1994.

[24] K. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1988.

[25] S. P. Amarasinghe, et al., "The SUIF compiler for scalable parallel machines," in *Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing*, 1995.

[26] M. Manjunathaiah and D. A. Nicole, "Advanced parallel usage analysis," in *First Int. Workshop on Parallel Processing*, 1994.

[27] M. Debbage, et al., "Southampton's portable occam compiler (SPOC)," in *WOTUG-17*, 1994.

[28] J. Grosch and H. Emmelmann, "A toolbox for compiler construction," GMD Karlsruhe, Germany, Tech. Rep. 20, 1990.

[29] B. Creusillet, "In and out array region analysis," in *Fifth Workshop on Compilers for Parallel Computers*, 1995.

[30] P. D. Hovland and L. M. Ni, "A model for automatic data partitioning," in *Int. Conf. Parallel Processing*, vol. 2, pp. 251–259, 1993.

[31] J. Stichnoth, "Efficient compilation of array statements for private memory multicomputers," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-93-109, 1993.

[32] S. Carr and K. Kennedy, "Compiler blockability of numerical algorithms," Department of Computer Science, Rice University, Houston, Tech. Rep. CRPC-TR92208-S, April 1992.

[33] U. N. Shenoy, et al., "An automatic parallelization framework for multicomputers," *Computer Languages*, vol. 20, pp. 135–150, 1994.