Symbolic Array Dataflow Analysis for Array Privatization and Program Parallelization [1]

Junjie Gu Zhiyuan Li Gyungho Lee†

Department of Computer Science
†Department of Electrical Engineering
University of Minnesota
200 Union Street S.E.
Minneapolis, MN 55455

{gu,li}@cs.umn.edu, ghlee@ee.umn.edu

Abstract

Array dataflow information plays an important role for successful automatic parallelization of Fortran programs. This paper proposes a powerful symbolic array dataflow analysis to support array privatization and loop parallelization for programs with arbitrary control flow graphs and acyclic call graphs. Our scheme summarizes array access information using **guarded array regions** and propagates such regions over a **Hierarchical Supergraph (HSG)**. The use of guards allows us to use the information in IF conditions to sharpen the array dataflow analysis and thereby to handle difficult cases which elude other existing techniques. The guarded array regions retain the simplicity of set operations for regular array regions in common cases, and they enhance regular array regions in complicated cases by using guards to handle complex symbolic expressions and array shapes. Scalar values that appear in array subscripts and loop limits are substituted on the fly during the array information propagation, which disambiguates the symbolic values precisely for set operations. We present efficient algorithms that implement our scheme. Initial experiments of applying our analysis to Perfect Benchmarks show promising results of improved array privatization.

Key words: Parallelizing compiler, array dataflow analysis, interprocedural analysis, array privatization, symbolic analysis.

1 Introduction

Recent experiments show that there exists significant performance difference between automatically and manually parallelized codes [7, 8]. One important factor causing such a discrepancy is related to array variables. Quite often, array elements written in one iteration of a DO loop are used in the same iteration before being overwritten in the next iteration. This kind of arrays usually serves as a temporary working space within an iteration and the array values in different iterations are unrelated. Using the same array for all iterations causes unnecessary loop-carried output and anti- dependences [20, 2] which prevent DO loops from being parallelized. Array privatization is a technique that creates a distinct copy of an array for each processor such that storage conflicts can be eliminated without violating program semantics. Table 1 summarizes the effect of array privatization on five Perfect benchmarking programs [22, 13, 12]. The fourth column of Table 1 shows the percentage of the sequential execution time (over the whole program) of each loop which is made parallel after array privatization. This percentage indicates the significance of each loop.

[1] This work is sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. This work is also supported in part by the National Science Foundation, Grant CCR-9210913 and a funding from Samsung Electronics.

Table 1: Summary of loops and privatization techniques

Program	Routine /Loop	Loop[1] Spdup	% of Seq	T1[2]	T2[2]	T3[2]
TRACK	nlfilt/300	5.2	40%	No	No	Yes
MDG	interf/1000	6.0	90%	Yes	Yes	Yes
	poteng/2000	5.2	8%	No	No	Yes
TRFD	olda/100	16.4	69%	Yes	No	No
	olda/300	12.3	29%	Yes	No	No
OCEAN	ocean/270	8.0	3%	Yes	Yes	Yes
	ocean/480	6.1	4%	Yes	Yes	Yes
	ocean/500	6.5	3%	Yes	Yes	Yes
ARC2D	filerx/15	4.0	7%	Yes	Yes	No
	filery/39	4.0	7%	Yes	No	No
	stepfx/300	3.0	21%	Yes	No	Yes
	stepfy/420	3.0	16%	Yes	No	Yes

1: Speedup is over sequential time. Speedups for ARC2D loops are estimates based on the maximal number of parallel iterations. Speedups for all others are measured on Alliant Fx/8 of 8 processors with vector units in each [13].
2: T1: Symbolic Analysis. T2: IF Condition Analysis T3: Interprocedural Analysis.

Array privatization requires a thorough analysis of array data flow and often involves the handling of routine calls. While the effects of IF conditions and CALL statements are shown to be important in practice [22, 37], no existing works handle such cases. The examples in Figure 1 illustrate such cases. In these three examples, privatizing the array A will make it possible to parallelize the I loops. Figure 1(a) shows a simplified version of a loop from the MDG program (routine interf) [6]. It is a difficult example which requires inferences between IF conditions. Although both A and B are privatizable, we will discuss A only, since B is a simple case. Suppose that the condition $(kc.NE.0)$ is false and, as the result, the last loop with index K within loop I gets executed and $A(6:9)$ gets used. We want to determine whether $A(6:9)$ may use values written in previous iterations of loop I. The condition $(kc.NE.0)$ being false implies that, within the same iteration of I, the statement $kc = kc+1$ is not executed. Thus, another condition $(B(K).GT.cut2)$ is false for all $K = 1, \cdots, 9$ of the first DO loop with index K. This fact further implies that the condition $(B(K+4).GT.cut2)$ is false for $K = 2, \cdots, 5$ of the second DO loop with index K, which ensures that $A(6:9)$ gets written before its use in the same iteration I. Therefore, A is privatizable in loop I.

Figure 1(b) illustrates a simplified version of a segment of the ARC2D program(routine filerx)[6]. Using existing algorithms[27, 22, 37], an use of $A(jmax)$ will be deemed as possibly upwards exposed [1, ch 10] in each iteration, and $A(jmax)$ is also deemed as possibly written in the previous iterations. Therefore the existing algorithms would assume a loop-carried flow dependence which prevents array A to be privatized. However, a closer examination reveals that the condition $(.NOT.p)$ is invariant for DO loop I. As the result, if $A(jmax)$ is not modified in one iteration, which exposes its use, $A(jmax)$ should not be modified in any iteration. Therefore, $A(jmax)$ never uses any value written in previous iterations of I. Moreover, it is easy to see that the use of $A(jlow:jup)$ is not upwards exposed. Hence, A is privatizable and loop I is a parallel loop. In this example, the IF condition being loop invariant guarantees that there are no loop-carried flow dependences.

Figure 1(c) shows a simplified version of a segment of the OCEAN program(routine ocean)[6]. Interprocedural analysis is needed in this case. In order to privatize A in the I loop, the compiler must recognize the fact that the use of array A must take the values defined in the same iteration of I because if a call to *out* in the I loop does use $A(1:m)$, then the call to *in* in the same iteration must modify $A(1:m)$. This is because the condition $(x > SIZE)$ in subroutine *out* implies the condition $(x > SIZE)$ in subroutine *in*. For all three examples above, it is also necessary for the compiler to manipulate symbolic expressions.

```
DO I=1, nmol1
   kc=0
   DO K=1,9
      B(K) = .
      IF(B(K).GT.cut2) kc=kc+1
   ENDDO
   ...
   DO K=2,5
      IF(B(K+4).GT.cut2) goto 1
      A(K+4) = .
1: ENDDO
   ...
   IF (kc.NE.0) goto 2
   DO K=11,14
      ttemp = ...A(K-5)..
   ENDDO
2:
ENDDO
```

(a)

```
DO I = 1, 4
   DO J = jlow, jup
      A(J) = .
   ENDDO
      ...
   IF (.NOT.p)
      A(jmax) = .
   ENDIF
      ...
   DO J = jlow, jup
      . = A(J) + A(jmax)
   ENDDO
ENDDO
```

(b)

```
DO I = 1, n
   x = ...
   call in(A, x, m)
   ...
   call out(A, x, m)
ENDDO
SUBROUTINE in(B, x, mm)
   IF (x>SIZE) RETURN
   DO J = 1, mm
      B(J) = .
   ENDDO

SUBROUTINE out(B, x, mm)
   IF (x>SIZE) RETURN
   DO J = 1, mm
      = B(J)
   ENDDO
END
```

(c)

Figure 1: Examples of Privatizable Arrays.

To summarize the above, a powerful compiler must handle IF conditions and perform symbolic analysis and interprocedural analysis. Table 1 lists information from several Perfect Benchmark programs, in which the last three columns show the techniques needed to privatize arrays within each loop. It is important to note that the difference between the handling of IF statements and the handling of IF conditions. In traditional dataflow analysis, IF statements may be handled by conservatively merging information from all branches without considering the contents of the IF conditions. Such a treatment is insufficient for the cases in Figure 1.

Previous works on array dataflow analysis can be divided into two categories. The first one attempts to gather flow information for each array element and to acquire an exact, complete array data flow information for all array elements. Feautrier [14] suggests to establish a *source function* for each array use to indicate which definitions will define the value for each distinct array element of the array use reference. Maydan et al [27, 28] simplify Feautrier's method by using a Last-Write-Tree(LWT). Duesterwald et al [11] compute the dependence distance for each reaching definition within a loop. Pugh and Wonnacott [32] use a set of constraints to describe array dataflow problems and solve them basically by Fourier-Motzkin variable elimination. Most works in this category so far do not handle IF statements, symbolic expressions, or routine calls because these complications make the computation of source functions, LWT's, dependence distances, or the operations on constraints more difficult. Recently, Maslov [26] extends the previous works in this category by handling certain IF conditions, but he restricts the program to be well structured and to have no multiple exits from structures, which is not the case in many practical programs. Maslov does not discuss how to legally propagate IF conditions for symbolic comparison. In the second category, array elements are not analyzed individually. Instead, a set of array elements of a regular shape, called a *regular array region* or *section*, is treated as a single unit. Simple set operations such as union, intersection, and difference, are performed on such units. Works by Gross and Steenkiste [16], Rosene [33], Li [22], Tu and Padua [37], and Granston and Veidenbaum [15], can roughly be included in this category. These works do not provide as many details about reaching-definitions as the first category. However, they handle more complex program constructs such as IF statements. Neither of those two categories, however, has taken the IF conditions into account except Tu and Padua's work, in which the handling of IF conditions is mensioned briefly and seems not to be in a systematic way. Hence, existing array dataflow algorithms are not sophisticated enough to handle practical programs such as those in Table 1.

In this paper, we present an interprocedural symbolic array dataflow analysis which takes not only IF statements into account , but also IF conditions. Our analysis belongs to the second category mentioned above in that we normally perform set operations on regular array regions such as rectangles, and we are not concerned about reaching-definitions for individual array elements. However, in order to take account of IF conditions under which an array reference is issued and to represent the set operation results in the presence of symbolic terms, we introduce a reference predicate which further qualifies a regular array region. Using such a *guarded array region*(GAR) to summarize array references during propagation is unique to our method of array dataflow analysis. The GAR's retain the simplicity of set operations for regular array regions in common cases, and they enhance regular array regions in complicated cases by using guards to handle complex symbolic expressions and array shapes. Scalar values that appear in array subscripts and loop limits are substituted on the fly during the array information propagation, which disambiguates the symbolic values precisely for set operations. Our analyzer handles Fortran programs in particular, but it can be extended to handle other imperative languages. The analyzer uses the array dataflow analysis results to privatize arrays and to parallelize DO loops.

The remaining of the paper is organized as follows. Section 2 presents the background. Section 3 discusses GAR operations and the applications of GAR's for array privatization and loop parallelization. Section 4 gives algorithms to collect and manipulate summary information. Section 5 discusses a few extensions and our current implementation. Section 6 reports our preliminary experiments with automatic array privatization. We conclude this paper in Section 7.

2 Background

In this section, we provide the background of this paper, especially the difference between conventional data dependence analysis and our array dataflow analysis.

Array dataflow analysis refers to computing the flow of values for array elements. It can be described as follows: given one or several use references of an array, find their reaching-definitions, i.e. the mod (modification) references which produce the values for those references; or reversely, given one or several mod references of an array, find the use references which consume the values written by those mod references. Array dataflow analysis can be at a very low level, analyzing reference by reference and even element by element. It can also be at a higher level, analyzing program segments rather than individual references. For array privatization and loop parallelization, analysis at the loop iteration level normally suffices.

Conventional data dependence analysis is the predecessor of all current works on array dataflow analysis. In his pioneering work, D.J. Kuck defines flow dependences, anti- dependences and output dependences [20]. While the latter two are due to multi-assignments in imperative languages, a flow dependence is defined between two statements, one of which reads the value written by the other. Thus, the original definition of flow dependences is precisely a reaching-definition relation. Nonetheless, early compiler techniques were not able to compute array reaching-definitions and therefore, for a long time, flow dependences are conservatively computed by asserting that one statement depends on another if the former may execute after the latter and both may access the same memory locations. Thus, the analysis of all three kinds of data dependences degenerates to the problem of memory disambiguation, which is insufficient for array privatization. There exist three main approaches to memory disambiguation. The first one, *numerical methods*, establishes algebraic equations between array subscripts and determines whether the equations are solvable subject to the loop limits and dependence directions [4, 5, 31, 24, 25, 19, 38]. Numerical methods normally do not apply to array subscripts and loop limits that contain unknown symbolic terms[34]. The second approach, originally proposed to handle call statements, uses array range triples to represent regular array regions which summarize the array elements accessed by one or several references [10, 17, 3, 9]. The third approach, which is the most general but also the most time consuming, represents the set of referenced array elements

by a set of inequalities and equations and uses Fourier-Motzkin pairwise elimination or integer programming to determine the feasibility of the set [21, 36, 35, 31]. The third approach can take IF conditions into account, while the other two do not. Pugh and Wonnacott [30] also discuss an extension of the Omega test [31] which computes certain array reaching-definitions for special cases without IF statements.

In this paper, we adopt a combination of the array range triple representation and the (in)equality representation to summarize array references. We use such summaries to perform array dataflow analysis which is much more powerful than conventional data dependence analysis.

3 Guarded array regions

In order to perform array privatization automatically, array reference information must be summarized for program segments. We adopt a format called guarded array regions(GAR's) for the summaries. A GAR contains a regular array region and a guard. In the following, we define our regular array regions first and then define GAR's.

Definition A *regular array region* of array A is denoted by $A(r_1, r_2, \cdots, r_m)$, where m is the dimension of A, r_i, $i = 1, \cdots, m$, is a range in the form of $(l : u : s)$, and l, u, s are symbolic expressions. The triple $(l : u : s)$ represents all values from l to u with step s. An empty array region is represented by $\emptyset$ and an unknown array region is represented by Ω.

The regular array region defined above is more restrictive than the regular sections used in the ParaScope environment at Rice University [10, 3, 17]. Where more complex array shapes arise, however, we can always add more information to the guards in GAR's to describe the shapes more precisely (more on this issue in Section 5), which we have not found necessary for array privatization in practice so far. The primary purpose of the guards, nonetheless, is to describe IF conditions under which regular array regions are accessed.

Definition A *guarded array region*(GAR) is a tuple $[P, R]$ which contains a *regular array region* R and a guard P, where P is a predicate that specifies the condition under which R is accessed. We use Δ to denote a guard whose predicate cannot be written explicitly, i.e. an unknown guard. If both $P = \Delta$ and $R = \Omega$, we say the GAR $[P, R] = \Omega$ is unknown. Similarly, if either P is $False$ or R is $\emptyset$, we say $[P, R]$ is $\emptyset$.

Note that if R contains symbolic terms, then the inequalities implied by the valid ranges of R are explicitly included in P. Thus, the emptiness of $[P, R]$ can be detected by examining P only. For any given program segment, we use GAR's to summarize the sets listed below. The side effect of a program segment can be captured completely by these mod sets and upwards exposed sets. Since we use guards, all these sets are exact sets unless the GAR's contain unknown components.

- UE – The set of the upwards exposed array elements which are used within this segment and take values defined outside this segment.
- UE_i – For an arbitrary iteration i of a DO loop, the set of the upwards exposed array elements which are used within this iteration and take values defined outside this iteration.
- MOD – The set of array elements written within this segment.
- MOD_i – For an arbitrary iteration i of a DO loop, the set of the array elements written within that iteration.
- $MOD_{<i}$ – For an arbitrary iteration i of a DO node, the set of the array elements written within the iterations prior to i.

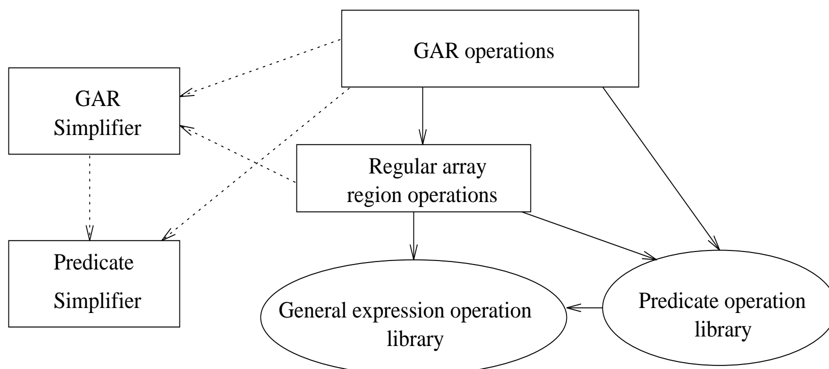


Figure 2: Overview of reference tuple and expression operations

- $MOD_{>i}$ – For an arbitrary iteration i of a DO node, the set of the array elements written within the iterations following i.

Take Figure 1(c) for example. For loop J of subroutine *in*, UE_j is empty and MOD_j equals $[True, B(j:j:1)]$ (or $[True, B(j)]$), therefore $MOD_{<j}$ is $[1 < j, B(1:j-1:1)]$ and $MOD_{>j}$ is $[j < mm, B(j+1:mm:1)]$. The set MOD for the loop J is $[1 \leq mm, B(1:mm:1)]$. Thus, the set MOD of subroutine *in* is $[x \leq SIZE \wedge 1 \leq mm, B(1:mm:1)]$. Similarly, UE_j for loop J of subroutine *out* is $[True, B(j:j:1)]$, and UE for the same loop is $[1 \leq mm, B(1:mm:1)]$. The set UE of the subroutine *out* is $[x \leq SIZE \wedge 1 \leq mm, B(1:mm:1)]$.

3.1 Operations on GAR's

The work in this paper requires three kinds of operations on GAR's, namely, union, intersection, and difference. These operations in turn are based on union, intersection, and difference operations on regular array regions as well as logical operations on predicates. Since symbolic terms may appear in both arithmetic expressions and predicates, we implement a GAR simplifier and a predicate simplifier to simplify GAR's and predicates. Figure 2 gives a diagram to show the components in our analyzer that handles GAR operations. The *general expression operation library* provides routines performing the operations of addition, subtraction, multiplication, and division with an integer constant divisor on integer symbolic expressions, which are normalized to an ordered sum of products. The *predicate operation library* provides routines which perform operations such as AND, OR, and NOT as well as the other logical operators in FORTRAN on predicates written in an ordered conjunctive normal form(CNF). The *GAR simplifier* simplifies GAR's and removes redundant ones. The *predicate simplifier* is used to determine whether a predicate in a GAR is false or true and to remove redundant predicate components. If symbolic expressions become too complex, e.g. if they contain multiplications of more than one index variable, then the described array regions are marked as unknown. We leave the discussion of the simplifiers to Section 5 and focus on the definitions of GAR operations and regular array regions in this subsection.

In most cases in practice, the results of GAR operations are quite simple. However, in general, we need to use a list of GAR's as a representation for the UE and MOD sets defined previously. An important note to make is that for the ranges in a regular array region, the requirement that the lower bound never exceeds the upper bound is always imposed explicitly by including this condition in the guard. (Where no confusion

results, however, we may omit such conditions in our representation in this paper just for simplicity.). This treatment allows the operations on range triples to proceed in a straightforward way without having to distinguish many different cases. As an example, consider two GAR's $T_1 = [a \le b, A(a : b : 1)]$, $T_2 = [b \le c, A(b : c : 1)]$. As we will define later, we have

$$\begin{aligned} T_1 \cup T_2 &= [a \le b \wedge b \le c, A(a : b : 1) \cup A(b : c : 1)] \\ &\quad \cup [a \le b \wedge b > c, A(a : b : 1)] \cup [a > b \wedge b \le c, A(b : c : 1)] \end{aligned}$$

where $A(a : b : 1) \cup A(b : c : 1)$ can be merged into $A(a : c : 1)$ without the concern of whether $a > b$ or $b > c$ is the case.

Regular array region operations: In general cases, the results of regular array region operations may be a list of several regular array regions instead of just one, possibly with new conditions produced. So, we use a list of GAR's to represent them.

In order to decompose regular array region operations into the operations on different array dimensions, it is convenient to use the notation of *guarded ranges* which impose conditions on symbolic terms which appear in range triples. For example, $[True, (1 : 10 : 1)]$ is a one-dimensional guarded range and $[c < d, (c : d : 1, c : d+1 : 1)]$ is a two-dimensional guarded range. Thus, a regular array region of a m-dimensional array A can be written as an m-dimensional guarded range. As operands of the region operations must belong to the same array, we will drop the array name from the array region notation hereafter whenever there is no confusion. Given two regular array regions, $R_1 = A(r_1^1, r_2^1, \cdots, r_m^1)$, $R_2 = A(r_1^2, r_2^2, \cdots, r_m^2)$, where m is the dimension of array A, we define the following operations:

- $R_1 \cap R_2$: For the sake of simplicity of presentation, here we assume steps of 1 and leave Section 5 to discuss other step values. Let $r_i^1 = (l_i^1 : u_i^1 : 1)$, $r_i^2 = (l_i^2 : u_i^2 : 1)$, $i = 1, \cdots, m$. Let D_i be $r_i^1 \cap r_i^2$, we have $D_i = [True, (max(l_i^1, l_i^2) : min(u_i^1, u_i^2) : 1)]$. We handle the max and min operations by replacing them with inequalities as shown in the following formula:

$$\begin{aligned} D_i = & [l_i^1 \le l_i^2 \wedge u_i^1 \le u_i^2, (l_i^2 : u_i^1 : 1)] \cup \\ & [l_i^1 \le l_i^2 \wedge u_i^1 > u_i^2, (l_i^2 : u_i^2 : 1)] \cup \\ & [l_i^1 > l_i^2 \wedge u_i^1 \le u_i^2, (l_i^1 : u_i^1 : 1)] \cup \\ & [l_i^1 > l_i^2 \wedge u_i^1 > u_i^2, (l_i^1 : u_i^2 : 1)] \\ = & \bigcup_{j=1,\cdots,4} [p_i^j, d_i^j] \end{aligned}$$

  where $p_i^j, d_i^j, j = 1, \cdots, 4$ are:

$$\begin{array}{ll} p_i^1 = l_i^1 \le l_i^2 \wedge u_i^1 \le u_i^2; & d_i^1 = (l_i^2 : u_i^1 : 1) \\ p_i^2 = l_i^1 \le l_i^2 \wedge u_i^1 > u_i^2; & d_i^2 = (l_i^2 : u_i^2 : 1) \\ p_i^3 = l_i^1 > l_i^2 \wedge u_i^1 \le u_i^2; & d_i^3 = (l_i^1 : u_i^1 : 1) \\ p_i^4 = l_i^1 > l_i^2 \wedge u_i^1 > u_i^2; & d_i^4 = (l_i^1 : u_i^2 : 1) \end{array}$$

  Then $R_1 \cap R_2$ equals:

$$\begin{cases} \emptyset & \exists i, D_i = \emptyset \\ \bigcup_{j_1, \cdots, j_m \in [1,4]} [p_1^{j_1} \wedge \cdots \wedge p_m^{j_m}, (d_1^{j_1}, \cdots, d_m^{j_m})] & Otherwise \end{cases}$$

  One should realize that in practice, the intersection is usually much simpler than the above general formula indicates, as many of the unioned components can be immediately recognized as empty. For example, let $r_i^1 = (a : 100 : 1)$ and $r_i^2 = (b : 100 : 1)$. We have $r_i^1 \cap r_i^2 = [a > b, (a : 100 : 1)] \cup [a \le b, (b : 100 : 1)]$. Note that we do not keep max and min operators in a regular array region. Instead, we replace them by explicit inequalities and place them in the guards, which makes it possible for our simplifiers to remove empty and redundant GAR's as early as possible.

- $R_1 \cup R_2$: If the union can be represented as one regular array region R_3, then the result is R_3. Otherwise, the result is a list of two regular array regions R_1, R_2. For example, $(1:a:1) \cup (a+1:100:1) = (1:100:1)$.

- $R_1 - R_2$: For an m-dimensional array, the general result of the difference operation is 2^m regular regions if each range difference results in two new ranges, whose representation could be quite complex for large m. However, it is useful to describe the general formulas of set difference operations. Suppose $R_1 \supseteq R_2$ (otherwise, use $R_1 - R_2 = R_1 - R_1 \cap R_2$). We first define $R_1(k)$ and $R_2(k)$, $k = 1, \cdots, m$, as the last k ranges within R_1 and R_2 respectively. We have $R_1(m) = (r_1^1, r_2^1, r_3^1, \cdots, r_m^1)$ and $R_2(m) = (r_1^2, r_2^2, r_3^2, \cdots, r_m^2)$, and $R_1(m-1) = (r_2^1, r_3^1, \cdots, r_m^1)$ and $R_2(m-1) = (r_2^2, r_3^2, \cdots, r_m^2)$, for example. The computation of $R_1 - R_2$ is recursively given by the following formula:

$$R_1(m) - R_2(m) = \begin{cases} (r_1^1 - r_1^2) & If\ m = 1 \\ (r_1^1 - r_1^2, r_2^1, r_3^1, \cdots, r_m^1) \cup (r_1^2, (R_1(m-1) - R_2(m-1))) & If\ m > 1 \end{cases}$$

  In general, $r_1^1 - r_1^2$ splits into two guarded ranges(see Section 5). Let them be $[p_1, d_1^1] \cup [p_2, d_1^2]$, where d_1^1 and d_1^2 are ranges. $R_1(m-1) - R_2(m-1)$ is in turn calculated by reusing the above formula. Assuming the final result of $R_1(m-1) - R_2(m-1)$ is

$$R_1(m-1) - R_2(m-1) = \bigcup_{j=1,\cdots,k} [P_j, (d_2^j, d_3^j, \cdots, d_m^j)],$$

  we then have,

$$\begin{array}{l} (r_1^1 - r_1^2) = [p_1, d_1^1] \cup [p_2, d_1^2] \\ (r_1^1 - r_1^2, r_2^1, r_3^1, \cdots, r_m^1) = [p_1, (d_1^1, r_2^1, r_3^1, \cdots, r_m^1)] \cup [p_2, (d_1^2, r_2^1, r_3^1, \cdots, r_m^1)] \\ (r_1^2, (R_1(m-1) - R_2(m-1))) = \bigcup_{j=1,\cdots,k} [P_j, (r_1^2, d_2^j, d_3^j, \cdots, d_m^j)] \end{array}$$

  The final result of $R_1 - R_2$ is equal to $R_1(m) - R_2(m)$ which is a list of m-dimensional guarded ranges. For example,
  $(1:100:1) - (a:30:1) = [1 < a, (1:a-1:1)] \cup [True, (31:100:1)]$.
  $(1:100:1, 1:100:1) - (20:30:1, a:30:1)$
  $= ((1:100:1) - (20:30:1), 1:100:1) \cup (20:30:1, (1:100:1) - (a:30:1))$
  $= [True, (1:19:1, 1:100:1)] \cup [True, (31:100:1, 1:100:1)]$
  $\cup [1 < a, (20:30:1, 1:a-1:1)] \cup [True, (20:30:1, 31:100:1)]$.

GAR operations: Since the result of regular region operations may be written as guarded ranges(see above), to facilitate the presentation of GAR operations, we find it convenient to use a notation $[[P, Tlist]]$, where $Tlist$ is a list of GAR's instead of a single regular array region. Given $Tlist = \bigcup_{j=1,\cdots,k}[P_j, R_j]$, $[[P, Tlist]]$ stands for $\bigcup_{j=1,\cdots,k}[P_j \wedge P, R_j]$. We call the notation $[[P, Tlist]]$ a *nested GAR*.

Given two GAR's, $T_1 = [P_1, R_1]$ and $T_2 = [P_2, R_2]$, we have the following:

- $T_1 \cap T_2 = [[P_1 \wedge P_2, R_1 \cap R_2]]$
- $T_1 \cup T_2 = [[P_1 \wedge P_2, R_1 \cup R_2]] \cup [P_1 \wedge \overline{P_2}, R_1] \cup [\overline{P_1} \wedge P_2, R_2]$
  The above formula can be simplified in the following three common cases:
  - If $P_1 \Rightarrow P_2$, the union becomes

$$[[P_1, R_1 \cup R_2]] \cup [\overline{P_1} \wedge P_2, R_2]$$

  - If $P_2 \Rightarrow P_1$, the union becomes

$$[[P_2, R_1 \cup R_2]] \cup [P_1 \wedge \overline{P_2}, R_1]$$

– If $R_1 = R_2$, the result is

$$[P_1 \vee P_2, R_1]$$

- $T_1 - T_2 = [[P_1 \wedge P_2, R_1 - R_2]] \cup [P_1 \wedge \overline{P_2}, R_1]$

The results of regular array region operations are often simple in practice and the emptiness of the result is easy to detect. Hence, GAR's retain the efficiency of regular array region operations in common cases, while enhancing the precision when necessary.

3.2 Applications of GAR's

3.2.1 Array Privatization

An array A is a privatization candidate in a loop L if its elements are overwritten in different iterations of L (see [22]). Such a candidacy can be established by examining the array subscripts: if the subscripts of array A do not contain any induction variables of L, then A is a candidate [22]. A privatization candidate is privatizable if there exist no loop-carried flow dependences in L. For an array A in a loop L with an index I, if $MOD_{<i} \cap UE_i = \emptyset$, then there exists no flow dependence carried by loop L. Take Figure 1(c) for example, A is obviously a privatization candidate. We have $UE_i = \emptyset$, $mod_{<i} = [x \leq SIZE \wedge 1 < m \wedge 1 < i, A(1:m:1)]$, and $MOD_{<i} \cap UE_i = MOD_{<i} \cap \emptyset = \emptyset$. So A is privatizable within loop I. As this example shows, the emptiness of the UE set can serve as a simple sufficient condition for privatizability.

Live analysis must be performed for privatized arrays in order to determine whether and which last values of the privatized arrays must be copied out of the DO loop. Previous works have addressed this issue [22, 37, 27].

3.2.2 Loop parallelization

The essence of loop parallelization is to prove the absence of loop-carried dependences. For a given DO loop L with index I, the existence of different types of loop-carried dependences can be detected in the following order:

1. **loop-carried flow dependences:** They exist if and only if $UE_i \cap MOD_{<i} \neq \emptyset$.
2. **loop-carried output dependences:** They exist if and only if $MOD_i \cap (MOD_{<i} \cup MOD_{>i}) \neq \emptyset$.
3. **loop-carried anti- dependences:** They exist if and only if $UE_i \cap MOD_{>i} \neq \emptyset$.

   This formula is valid in the absense of loop-carried output dependences. It is applied only after our algorithm successfully proves the absence of loop-carried flow and loop-carried output dependences in step 1 and 2. (If loop-carried anti- dependences are considered separately, they should be detected using DE_i instead of UE_i in the above formula, where DE_i is the *downwards exposed* use set of iteration i.)

Although the formulas above resemble those in previous works, e.g. [10], previous works do not use the flow-sensitive sets and thus are less precise.

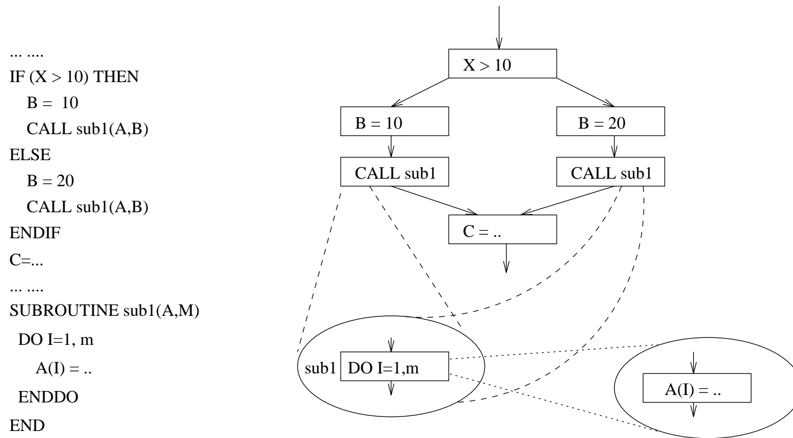


Figure 3: Example of the HSG.

4 Algorithms for symbolic array dataflow analysis

In this section, we present algorithms to calculate the MOD and UE information by propagating the GAR's over a *hierarchical supergraph* (HSG). The HSG in this paper is an enhancement of Myers' *supergraph* [29] which is a composition of the flow subgraphs of all routines in a program. In a supergraph, each call statement is represented by a node, termed a *call node* in this paper, which has an outgoing edge pointing to the entry node of the flow subgraph of the called routine. The call node also has an incoming edge from the unique exit node of the called routine. To facilitate the information summary for DO loops, we add a new kind of nodes, the *loop nodes*, to represent DO loops. The resulting graph, which we call the hierarchical supergraph (HSG), contains three kinds of nodes — basic blocks, loop nodes and call nodes. An IF condition itself forms a single basic block node. A loop node is a compound node which has its attached flow subgraphs describing the control flow within the DO loop. Due to the nested structures of DO loops and routines, a hierarchy is derived among the HSG nodes, with the flow subgraph at the highest level representing the main program. The HSG resembles the HSCG used by the PIPS project [18]. Figure 3 and Figure 5 show two HSG examples. Note that the flow subgraph of a routine is never duplicated for different calls to the same routine unless the called routine is duplicated to enhance its potential parallelism. We assume that the program contains no recursive calls. For simplicity of presentation, we further assume that a DO loop does not contain GOTO statements which make premature exits. We also assume that the HSG contains no cycles due to backward GOTO statements. Our implementation, however, does take care of multiple exits in DO loops and backward GOTO statements, making conservative estimates when necessary(see Section 5). In the flow subgraph of a loop node, the back edge from the exit node to the entry node is deliberately deleted, as it conveys no additional information for array summaries. Under the above assumptions and treatment, the HSG is a hierarchical *dag* (directed acyclic graph). The following subsection presents the information summary algorithms.

4.1 Summary Algorithms

As listed in Section 3, the summary information of our interest has two main kinds, the MOD summary and the UE summary. One unique aspect of this paper is that, because we attach guards to regular array regions, the calculation of the MOD information involves only union operations. (One needs to carefully distinguish our MOD summary from the conventional *kill set* summary which requires the intersection of the MOD

sets at the IF statements because the conventional summary does *not* include guards.) The calculation of the UE information, on the other hand, requires all three kinds of set operations.

We present algorithms for computing the MOD and UE information in this subsection. As the whole computation involves many delicate details, it is necessary to leave the nonessential ones out of the discussion due to the space limit. Throughout the remaining of the text, wherever no confusion results, we refer to the program segment represented by a HSG node by the node itself.

We first define an *expansion* function which is invoked by the information summary algorithms.

Expansion

For a loop with index i, where $l \leq i \leq u$, and a GAR, T, if T does not contain i in its representation, then the *expansion* of T by i is T itself. If T contains i in its representation, then the *expansion* of T by i is a GAR obtained by the following steps:

- If i appears in the guard of T, then i should be solved from the guard which, in general, is written as $l' \leq i \leq u'$, where l' and u' may be symbolic expressions. We obtain new bounds on i which is $max(l', l) \leq i \leq min(u', u)$. The inequalities and equalities involving i in the guard are then deleted.
- If i appears in only one dimension of T and the result of substituting $l \leq i \leq u$, or the new bounds on i obtained above, into the old range in that dimension can still be represented by a range $(l'' : u'' : s'')$, then we replace the old range by $(l'' : u'' : s'')$. Recall that the condition of $l'' \leq u''$ is placed in the guard and must be checked whenever necessary.
- If, in the above, the result of substitution of $l \leq i \leq u$ into the old range can no longer be represented by a range, then mark that dimension as Ω (unknown).
- If i appears in more than one dimension of T, then these dimensions are marked as Ω.

As an example, suppose a DO loop indexed by i has loop bounds $a \leq i \leq b$. Further suppose the given GAR is $T = [c \leq i + 1 \leq d, (1 : i : 1)]$. As the result of solving i from the guard, we have new bounds on i which are $max(a, c - 1) \leq i \leq min(b, d - 1)$. The expansion of T by i is $[True, 1 : min(b, d - 1) : 1)]$. The max and min operations are then replaced by explicit conditions as described in Section 3.

We now present our algorithms for information summary propagation. The algorithms *SUM_segment*, *SUM_call*, and *SUM_loop* are indirectly recursive. Figure 5 shows a complete example to illustrate the steps of the following algorithms. For simplicity, these algorithms are applied to one array only (In concept, the summary of more than one array can be acquired by applying these algorithms once for each of these arrays. In practice, the algorithms summarize all arrays at the same time).

SUM_segment: The algorithm for computing MOD and UE for a flow subgraph

Let $UE(n)$ and $MOD(n)$ be the upwards exposed use set and be the mod set for node n respectively, and let $UE_IN(n)$ and $MOD_IN(n)$ be the upwards exposed use set and mod set at the entry point of node n respectively. The algorithm is given below:

```
SUM_segment(mod, ue, G(s,e))
/* G(s,e): flow subgraph with starting node s and existing node e. */
/* mod is the mod set of G(s,e). */
/* ue is the upwards exposed use set of G(s,e). */
Step 1: find UE(n) and MOD(n) for each node n in G(s,e).
    FOR each node n in G(s,e) DO
        IF (n is a basic block)
            SUM_bb(mod(n), ue(n), n);
```

```
        ELSE IF (n is a loop node)
            SUM_loop(mod(n), ue(n), n);
        ELSE IF (n is a call node)
            SUM_call(mod(n), ue(n), n);
    ENDFOR
```

Step 2: Propagate mod and ue of each node backward, from e to s.

$mod_in(n) = mod(n) \cup (\bigcup_{p \in succ(n)} mod_in(p))$

$ue_in(n) = ue(n) \cup (\bigcup_{p \in succ(n)} ue_in(p) - mod(n))$

(Note that succ(e) = $\emptyset$.)

IF (n is a basic block containing IF-condition)
  add the condition to the guard of each GAR in $mod_in(n)$ and $ue_in(n)$

IF any expression in the $mod_in(n)$ and $ue_in(n)$ contains a variable that is defined within n, then that variable must be substituted by the right-hand-side of the defining statement within n. If the right-hand-side is too complicated, the expression is marked as unknown. If a variable is defined by a procedure or a function, we propagate information through the subgraph of this procedure or function.

At the end of the propagation, we have $mod = mod_in(s)$, $ue = ue_in(s)$.

SUM_bb: The algorithm for a basic block

For each basic block, the following algorithm calculates the UE and MOD sets. Note that, in the beginning, each array region in the basic block represents a single array element.

```
 SUM_bb(mod, upexp, n)
/* n is the basic block node */
/* SUM_bb gives mod and upexp sets */

     upexp = ∅; mod = ∅;
     FOR each use u of A, DO
         tmp = [True, u];
         FOR each mod m of A such that m is prior to u, DO
             tmp = tmp - [True, m];
         ENDFOR
         upexp = upexp ∪ tmp;
     ENDFOR
     /* calculate mod */
     FOR each mod m of A, DO
         mod = mod ∪ [True, m];
     ENDFOR
```

Algorithm for a call node

```
 SUM_call(mod, ue, n)
/* n is a call node */
/* mod is the mod set of n. */
/* ue is the upwards exposed set of n. */
      Let G(s,e) be the called subroutine
      1.  SUM_segment(mod, ue, G(s,e));
      2.  Map real parameters to the formal parameters in mod and ue.
```

Algorithm for a loop node

```
 SUM_loop(mod, ue, n)
/* n is a loop node */
/* mod is the mod set of n. */
/* ue is the upwards exposed set of n. */
      Let G(s,e) be the subgraph for this loop body.
      1. SUM_segment(mod_i, ue_i, G(s,e));
      2.1 Calculate the upwards exposed set:
            Compute mod_<i;
            ue_i_out = ue_i − mod_<i;
            /* expand the ue_i_out */
            ue = expand ue_i_out;
      2.2 Calculate the mod set:
            mod = expand mod_i;
```

5 Extensions and implementation details

In this section, we provide several details and possible extensions which we left out in previous sections. We also address our current implementation.

5.1 The range operations

In Section 3, we assumed step values of one in the discussion of range intersection. Here, we give a complete discussion of our treatment for other step values.

To describe the range operations, we use the functions of $min(e_1, e_2)$ and $max(e_1, e_2)$ in the following. However, these functions are removed by the technique stated in Section 3.

Given two ranges r_1 and r_2, $r_1 = (l_1 : u_1 : s_1)$, $r_2 = (l_2 : u_2 : s_2)$.

1. If $s_1 = s_2 = 1$,
   - $r_1 \cap r_2 =$ $[max(l_1, l_2) \leq min(u_1, u_2), (max(l_1, l_2) : min(u_1, u_2) : s_1)]$
   - Assuming $r_2 \subseteq r_1$ (otherwise use $r_1 - r_2 = r_1 - r_1 \cap r_2$), we have $r_1 - r_2 = [l_1 < max(l_1, l_2), (l_1 : max(l_1, l_2) - 1 : s_1)] \cup [min(u_1, u_2) < u_1, (min(u_1, u_2) + 1 : u_1 : s_1)]$
   - Union operation. If $(l_2 > u_1 + s_1)$ or $(l_1 > u_2 + s_2)$, $r_1 \cup r_2$ cannot be combined into one range. Otherwise, $r_1 \cup r_2 = [True, (min(l_1, l_2) : max(u_1, u_2) : s_1)]$
2. If $s_1 = s_2 = c > 1$, where c is a known constant value, we do the following: If $(l_1 - l_2)$ is divisible by c, then we use the formulas in case 1 to compute the intersection, difference and union. Otherwise, $r_1 \cap r_2 = \emptyset$ and $r_1 - r_2 = r_1$. The union $r_1 \cup r_2$ usually can not be combined.
3. If $s_1 = s_2$ and $l_1 = l_2$ (which may be symbolic expressions), then we use the formulas in case 1 to perform the intersection, difference and union.
4. If s_1 is divisible by s_2, we check if r_2 covers r_1. If so, we have $r_1 \cap r_2 = r_1$ and $r_1 \cup r_2 = r_2$. For other cases, we divide r_2 into several smaller ranges with step s_1 and then apply the above formulas.
5. Otherwise, the results of intersection and difference are marked as unknown and the union keeps to be a list of two ranges.

5.2 The GAR simplifier and the predicate simplifier

The GAR simplifier is the top level simplifier which calls the predicate simplifier. It examines GAR's to eliminate redundant ones, and it combines several GAR's into one if possible. Our analyzer invokes the GAR simplifier whenever there are changes to GAR's during the summary propagation.

A predicate is represented by the Conjunctive Normal Form such as $C_1 \wedge C_2 \wedge \cdots \wedge C_n$, where each C_i, called a disjunction, which in turn is represented as $re_1 \cup re_2 \cup \cdots \cup re_m$. Each re_i is a relational expression in the form of either $(e\ op\ 0)$ or $(lvar\ op1\ val)$, where $lvar$ is a logical variable, $op1$ is either $=$ or $\neq$, and val is True or False. The expression $(e\ op\ 0)$ is a relational expression, where e stands for an arithmetic expression and op is a relational operator, $<$, $=$, or $\neq$. The other relational operators can be easily transformed to a combination of these three operators.

The predicate simplifier is a key part in the handling of predicates. It is invoked by the GAR simplifier or by GAR operations whenever there are changes to predicates. There exist powerful, but rather time-consuming, general methods, such as integer programming, for predicate simplification [30]. Currently, for the sake of efficiency, we implement a limited simplifier which evaluates the truth value of the conjunction of two disjunctions or the disjunction of two relational expressions. CNF's of arbitrary lengths, as defined in the last paragraph, are handled by feeding one pair of disjunctions (or relational expressions) to the simplifier at a time. Similarly, the simplifier removes redundant components by examining two disjunctions or two relational expressions at a time. Simplification involving more than two operands simultaneously is not implemented yet. This limited simplifier seems to suffice in our experiment so far. However, we plan to experiment with a more powerful simplifier if the need arises.

IF conditions handled in our implementation do not contain array references. Our implementation can process IF conditions which contain scalars only. We handle integer conditions more thoroughly than floating point ones. For induction variables, we first convert them to expressions of index variables, so they will fit our implementation model. We have not implemented existential and universal qualifiers $\exists$ and $\forall$. The example in Figure 1(a), however, does need this kind of qualifiers to guard the regular array regions. Reduction systems in the Artificial Intelligence area can certainly handle these more complicated predicates.

5.3 Regular array regions and GAR's

Regular array regions defined in Section 3 are rectangular regions in which different dimensions are independent. To represent nonrectangular regions, our GAR can be extended by introducing a special dimension symbol ψ_i for each dimension i. For example, the array diagonal $A(i,i), i = 1, \cdots, n$, can be represented as a GAR $[\psi_1 = \psi_2, A(1:n:1, 1:n:1)]$. Similarly, an upper triangle of array A, such as $A(i,j), i = 1, \cdots, n$, $j = i, \cdots, n$, becomes a GAR $[\psi_1 \leq \psi_2, A(1:n:1, 1:n:1)]$. A predicate, therefore, may contain two kinds of conditions, one for restricting array regions and the other for guarding array regions as stated in previous sections. Our experience with array privatization so far has not required such an extension.

If a GAR or predicate operation involves an unknown operand, the result in general is unknown. However, unknown GAR's and predicates can be removed in certain cases. For example, suppose MOD_1 contains all elements of array A and MOD_2 is Ω, then $MOD_1 \cup MOD_2 = MOD_1$. As an example for predicate operations, suppose $P_1 = \Delta$. Obviously $P_1 \vee True = True$ and $P_1 \wedge False = False$. We have implemented such special cases.

Table 2: Experimental Results on Loops with Privatizable Arrays

Program	Routine /Loop	Array Names	Status*
TRACK	nlfilt/300	P1,P2,P,PP1,PP2,PP,XSD	yes
MDG	interf/1000	RS,FF,GG,XL,YL,ZL	yes
		RL	no
	poteng/2000	RS,RL,XL,YL,ZL	yes
TRFD	olda/100	XRSIQ,XIJ	yes
	olda/300	XIJKS,XKL	yes
OCEAN	ocean/270	CWORK	yes
	ocean/480	CWORK, CWORK2	yes
	ocean/500	CWORK	yes
ARC2D	filerx/15	WORK	yes
	filery/39	WORK	yes
	stepfx/300	WORK	yes
	stepfy/420	WORK	yes

*: Status shows whether these privatizable arrays can be automatically privatized now.

5.4 Goto statements

In Section 4, we assumed there exist no backward goto's and no premature exits out of DO loops. Certain cycles due to backward goto's can be transformed to DO loops which can then be covered by our model in Section 4. For the other cycles due to backward goto's, we first condense them into condensed nodes in our HSG. Thus the resulting graph is still a dag. The GAR's for condensed nodes are conservatively approximated. For loops with premature exits due to goto's, we treat loop variant and loop invariant GAR's differently. The former are approximated by marking the loop variant components in the GAR's as unknown. The latter are propagated precisely by following the exit edges. Since these are special cases, we do not describe more details.

6 Preliminary experimental results

We have implemented our symbolic, interprocedural array dataflow analysis in our prototyping parallelizing analyzer, Panorama, and collected preliminary results for array privatization from some of the Perfect benchmark programs. The array dataflow analysis is built upon the interprocedural scalar reaching-definition chains and the Hierarchical Supergraph[23]. Several conventional data dependence tests are also implemented. The more expensive array dataflow analysis is applied only to loops whose parallelizability cannot be determined by the conventional data dependence tests. The preliminary results are shown in Table 2 and Figure 4. The last column of Table 2 presents the current status of array privatization achieved by Panorama. Since the current implementation cannot handle subscripts containing subscript variables, i.e. array elements, we replace subscript variables by their equivalent scalar expressions through forward substitution by hand if such expressions exist and are needed. For example, suppose $A[JM[I]]$ is an array reference and the integer array JM is defined as $JM[I] = I - 1$. We replace $JM[I]$ with $I - 1$ so that the reference to A becomes $A[I-1]$. Such a case happens in ARC2D which uses integer arrays JPLUS, JMINUS in the subscript expressions. All arrays listed in Table 2 can be privatized under our scheme except array RL in the MDG program. This case was illustrated in Figure 1(a), where array A is a pseudo name for array RL. Our implementation is currently unable to deal with this difficult case and the reason has been given in Section 5.2.

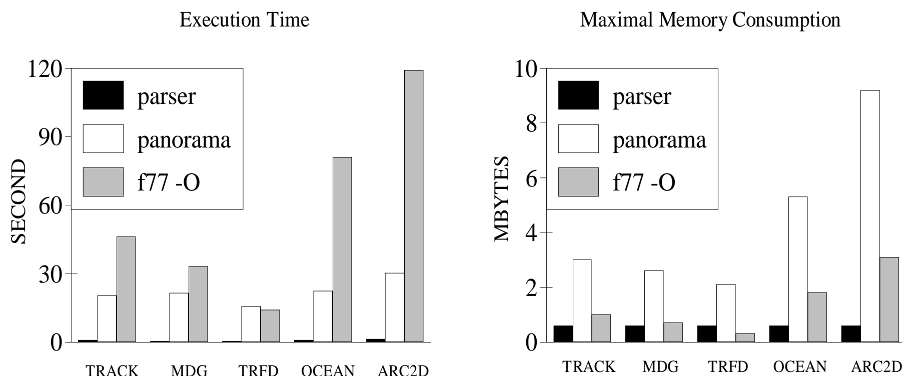


Figure 4: Comparison between Panorama and F77 concerning resource utilization (The 'panorama' bar includes the parser, conventional loop parallelization algorithms, and our array dataflow analysis in Panorama. The 'parser' bar includes the the Panorama parser only.)

At the time of writing this paper, the Panorama compiler does not generate parallel FORTRAN source code for any specific machine, although work is underway for Silicon Graphics power challenges. Meanwhile, we mark parallel loops internally. Thus, no speedup data is available yet. On the other hand, since the complexity of a sophisticated analysis as ours is of an important general concern, we provide data regarding the panorama's execution time and memory requirement.

Figure 4 gives a comparison between Panorama and FORTRAN compiler F77 regarding the elapsed time and the used memory. F77 is chosen for comparison because it is familiar to most readers. On the other hand, other research prototypes that perform analysis similar to ours have not published their execution time and memory requirement yet. A comparison with F77 gives a good indication whether our analysis is practical. Execution times shown for both Panorama and F77 are taken from the executions on Sun Sparc 2. The running time of Panorama is shorter than F77 with option -O, which suggests that the time spent by our analyzer is quite acceptable. However, the maximal memory utilization of Panorama is larger than that of F77 because the array summary information and interprocedural scalar information can occupy quite a large amount of memory.

7 Conclusion

Array dataflow information plays an important role for successful automatic program parallelization. However, existing techniques are not able to perform array dataflow analysis well because of the difficulty in handling interprocedural array dataflow, IF conditions, and symbolic expressions, resulting significant difference between automatically parallelized codes from manually parallelized ones.

In this paper, we have proposed a powerful interprocedural symbolic array dataflow analysis to support array privatization and program parallelization based on a Hierarchical Supergraph(HSG) and guarded array regions. Symbolic processing is integrated into the process of propagating array access information. The use of guards allows our analysis to handle IF conditions and to process symbolic terms efficiently.

Preliminary results for array privatization suggest that the prospect of efficient implementation of our powerful analysis is quite promising. Global privatization results can be obtained efficiently both in the amount of analysis time and in the amount of required memory. Our future work will focus on developing a more powerful symbolic manipulator and on improving our dataflow analyzer.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[2] J. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1984.

[3] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[4] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[5] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1993.

[6] M. Berry, D. Chen, P. Koss, D. Kuck, and S. Lo. The Perfect club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, University of Illinois, Urbana, IL, May 1989.

[7] Blume and Eigenmann. Symbolic analysis techniques needed or the effective parallelization of Perfect benchmarks. Technical report, Dept. of Computer Science, University of Illinois, 1994.

[8] W. Blume and R. Eigenman. Performance analysis of parallelizing compilers on the Perfect benchmarks programs. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[9] William Blume and Rudolf Eigenmann. The range test: A dependence test for non-linear expressions. Technical Report CSRD-Report-1345, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, April 1994.

[10] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *ACM SIGPLAN '86 Symp. Compiler Construction*, pages 162–175, June 1986.

[11] E. Duesterwald, R. Gupta, and M.L. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 68–77, June 1993.

[12] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua. Experience with Fortran program restructuring for Cedar multiprocessor. *Concurrency – Experience and Practice*, 5(7):553–573, October 1993.

[13] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect benchmark programs. In *Lecture Notes in Computer Science, 589.* Springer-Verlag, 1992.

[14] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 2(1):23–53, February 1991.

[15] E.D. Granston and A.V. Veidenbaum. Detecting redundant accesses to array data. In *Supercomputing '91*, November 1991.

[16] T. Gross and P Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software – Practice and Experience*, 20(2):133–155, February 1990.

[17] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. on Parallel and Distributed Systems*, 2(3), 1991.

[18] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *International Conference on Supercomputing*, pages 244–251, 1991.

[19] X. Kong, D. Klappholz, and K. Psarris. The I test: An improved dependence test for automatic parallelization and vectorization. *IEEE Trans. on Parallel and Distributed Systems*, 2(3), 1991.

[20] D.J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley & Sons, 1978.

[21] R. Kuhn. *Optimization And Interconnection Complexity for: Parallel Processors, Single-Stage Networks, And Decision Trees.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign,, Feb 1980.

[22] Z. Li. Array privatization for parallel execution of loops. In *ACM Int. Conf. on Supercomputing*, pages 313–322, July 1992.

[23] Z. Li. Propagating symbolic relations on an interprocedural and hierarchical control flow graph. Technical Report CSci-93-87, Computer Science Department, University of Minnesota, Minneapolis, MN, 1993.

[24] Z. Li and P.-C. Yew. Practical methods for exact data dependence analysis. In *Languages and Compilers for Parallel Computing*, pages 374–401. D. Gelernter, A. Nicolau, and D. Padua (Editors), MIT Press, 1991.

[25] Z. Li, P.-C. Yew, and C.-Q Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, January 1990.

[26] Vadim Maslov. Lazy array data-flow dependence analysis. In *Proceedings of Annual ACM Symposium on Principles of Programming Languages*, pages 331–325, Jan. 1994.

[27] D.E. Maydan, S.P. Amarasinghe, and M.S. Lam. Array data-flow analysis and its use in array privatization. In *Proc. of the 20th ACM Symp. on Principles of Programming Languages*, pages 2–15, January 1993.

[28] Dror E. Maydan. *Accurate Analysis of Array References.* PhD thesis, Stanford University, October 1992.

[29] E. W. Myers. A precise interprocedural data-flow algorithm. In *Proceedings of 8th Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, Jan. 1981.

[30] W Pugh and D Wonnacott. Eliminating false data dependences using the omega test. In *ACM SIGPLAN Conf. on Programming Languages Design and Implementation*, pages 140–151, June 1992.

[31] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing'91*, November 1991.

[32] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag.

[33] Carl Rosene. Incremental dependence analysis. Technical Report CRPC-TR90044, PhD thesis, Computer Science Department, Rice University, March 1990.

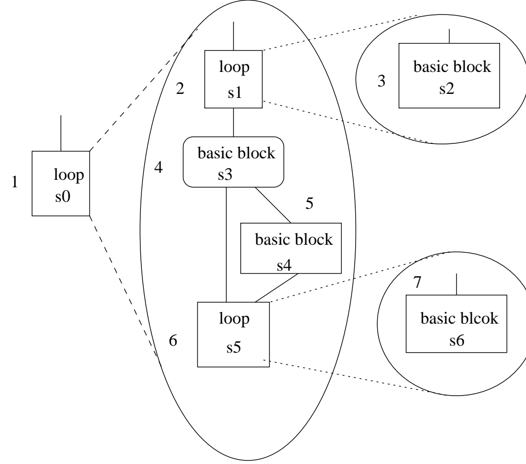
[34] Z. Shen, Z. Li, and P.-C. Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[35] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statments. In *ACM SIGPLAN'86 Sym. on Compiler Construction*, pages 176–185, July 1986.

[36] Remi Triolet. Interprocedural analysis for program restructuring with parafrase. Technical Report CSRD Rpt. No.538, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1985.

[37] Peng Tu and David Padua. Automatic array privatization. In *Proceedings of Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, August 1993.

[38] M.J. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *Int. Journal of Parallel Programming*, 16(2):137–178, April 1987.



A. $\boldsymbol{ue_i(1) = ?}$, $\boldsymbol{mod_i(1) = ?}$

1. Compute mod and ue for each nodes.
   $ue(2) = \emptyset;\ mod(2) = [T, (jlow : jup : 1)]$
   $ue(4) = mod(4) = \emptyset$
   $ue(5) = \emptyset;\quad mod(5) = [T, (jmax)]$
   $ue(6) = [T, (jlow : jup : 1)] \cup [T, (jmax)];\quad mod(6) = \emptyset$
2. Propagate information up.
   $ue_in(6) = ue(6);\quad mod_in(6) = mod(6)$
   $ue_in(5) = [jmax < jlow \vee jmax > jup, (jlow : jup : 1) \cup$
   $\qquad [jlow \leq jmax \leq jup, (jlow : jmax - 1 : 1) \cup (jmax + 1 : jup : 1)]$
   $mod_in(5) = [T, (jmax)]$
   $ue_in(4) = [P, (jlow : jup : 1)] \cup [P, (jmax)] \cup$
   $\qquad [\overline{P} \wedge (jmax < jlow \vee jmax > jup), (jlow : jup : 1)] \cup$
   $\qquad [\overline{P} \wedge (jlow \leq jmax \leq jup), (jlow : jmax - 1 : 1) \cup (jmax + 1 : jup : 1)]$
   $\quad = [P, (jmax)] \cup [P \vee jmax < jlow \vee jmax > jup, (jlow : jup : 1)] \cup$
   $\qquad [\overline{P} \wedge (jlow \leq jmax \leq jup), (jlow : jmax - 1 : 1) \cup (jmax + 1 : jup : 1)]$
   $mod_in(4) = [\overline{P}, (jmax)]$
   $ue_in(2) = [P \wedge (jmax < jlow \vee jmax > jup), (jmax)]$
   $mod_in(2) = [T, (jlow : jup : 1)] \cup [\overline{P}, (jmax)]$
   $ue_i(1) = ue_in(2);\quad mod_i(1) = mod_in(2)$

B. Is array A privatizable ?

$mod_{<i}(1) = [i > 1, (jlow : jup : 1)] \cup [i > 1 \wedge \overline{P}, (jmax)]$

$ue_i \cap mod_{<i}(1) = [P \wedge (jmax < jlow \vee jmax > jup) \wedge i > 1, (jlow : jup : 1) \cap (jmax)] \cup$
$\qquad [P \wedge (jmax < jlow \vee jmax > jup) \wedge i > 1 \wedge \overline{P}, (jmax)]$
$\qquad = \emptyset \longrightarrow A$ is privatizable

Figure 5: Privatizing array A in the example of Figure 1(b).