Interprocedural Analysis Based on Guarded Array Regions

Zhiyuan Li[1], Junjie Gu[1], and Gyungho Lee[2]

[1] Department of Computer Science, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455, USA
[2] Electrical Engineering, Division of Engineering, University of Texas - San Antonio, 6900 North Loop 1604 West, San Antonio, Texas 78249-0665, USA

Summary. Array data flow information plays an important role for successful automatic parallelization of Fortran programs. This chapter proposes a powerful symbolic array data flow summary scheme to support array privatization and loop parallelization for programs with arbitrary control flow graphs and acyclic call graphs. Our approach summarizes array access information interprocedurally, using **guarded array regions**. The use of guards allows us to use the information in IF conditions to do path-sensitive data flow summary and thereby to handle difficult cases. We also provide a mechanism to overcome the disadvantages of non-closed union and difference operations. This improves not only the exactness of summaries, but also the efficiency of the summarizing procedure. Our preliminary result on array privatization shows that our summary scheme is fast and powerful.

Key words: Parallelizing compilers, array data flow analysis, interprocedural analysis, array privatization, guarded array regions, symbolic analysis.

1. Introduction

Interprocedural analysis, i.e. analyzing programs across routine boundaries, has been recognized as an essential part of program analysis. Parallelizing compilers, as a tool to enhance program efficiency on parallel computers, depend heavily upon this technique to be effective in practice.

Interprocedural analysis for scalars has been well studied. In contrast, interprocedural analysis for array references is still an open issue. Currently, there are two ways to extend program analysis across a call site. The first and simpler way is to use inline expansion. The drawbacks of inlining are twofold. First, even if the program size does not increase after inline expansion, the loop body containing routine calls may grow dramatically. This often results in much longer compile time and much larger consumption of memory resources [21] because many compiler algorithms dealing with loops have non-linear complexity with respect to the loop body's size. Because a routine is analyzed each time it is inlined, duplicate analysis can also cause parallelizing compilers to be inefficient. Second, some routines are not amenable to inline expansion due to complicated array reshaping and must be analyzed without inlining.

These drawbacks of inlining make a *summary* scheme often a more desired alternative. Early works on summary schemes summarize the side effects of a called routine with sets of array elements that are modified or used by routine calls, called MOD sets and USE sets respectively. Data dependences involving routine calls can be tested by intersecting these sets. Existing approaches can be categorized according to methods of set representation. Convex regions [27, 28] and data access descriptors [2] define sets by a system of inequalities and equalities, while bounded regular sections [5, 15] use range tuples to represent sets. Even though bounded regular sections are less precise than convex regions and data access descriptors, they are much simpler and are easy to implement.

The commonalities of these previous methods are as follows. First, because they are path-insensitive, IF conditions are not taken into account when program branches are handled. Second, they are flow-insensitive, so only MOD/USE sets of array elements, which are modified or used respectively, are summarized. Third, they use a single descriptor (a single regular section, a single convex region, etc.) to summarize multiple references to the same array. Because union operations are not closed, approximations have to be made in order to represent union results in a single convex region or a single regular section. This causes the loss of summary information. Therefore, these methods are insufficient for optimization such as array privatization.

Recently, flow-sensitive summary, or array data flow summary, has been a focus in the parallelizing compiler area. The most essential information in array data flow summary is the upward exposed use (UE) set [7, 29, 17, 14, 12]. Our work [12] and that of M. Hall, et al. [14] use either a list of regular sections or a list of convex regions to summarize each array in order to obtain more precise information than that provided by a single regular section or convex region. Our work is unique in its use of *guarded array regions* (GAR's), providing the conditions under which an array region is modified or upward-exposedly used. This is in contrast to path-insensitive summaries which do not distinguish summary sets for different program paths. Compared to other approaches [7] which handle more restricted cases of IF conditions, our approach seems to be more general.

In this chapter, we will describe our array data flow summary based on guarded array regions in the context of parallelizing compilers. The techniques involved should also be applicable to various programming supporting systems which deal with arrays, e.g. compilers of explicit parallel programs. The remainder of the chapter is divided into the following sections. In section 2, we present brief background information. Section 3 discusses guarded array regions, followed by our summary algorithm in section 4. We address the implementation issues in section 5. The preliminary results using GAR's for array privatization are shown in section 6. Section 7 briefly discusses the related works. Finally, we conclude this chapter in section 8.

2. Preliminary

In this section, we provide background information regarding array reference summary in the context of data dependence analysis and of data flow analysis.

2.1 Traditional Flow-Insensitive Summaries

Data dependence analysis is widely used in program analysis and is a key technique for program optimization and parallelization. It is also a main motivation for interprocedural analysis. Traditionally, data dependence analysis concerns whether two statements modify and use a common memory location. Given two statements, $S1$ and $S2$, which use and modify array A, we assume $S1$ is executed before $S2$. Let MOD_s be the set of all elements of A that are modified by S and USE_s the set of all elements of A that are used by S. The dependences between S_1 and S_2 can be determined by the following set operations [30]:

- flow dependence: iff $MOD_{s1} \cap USE_{s2} \neq \emptyset$.
- anti- dependence: iff $MOD_{s2} \cap USE_{s1} \neq \emptyset$.
- output dependence: iff $MOD_{s1} \cap MOD_{s2} \neq \emptyset$.

When S_1, S_2, or both, are call statements, the above sets need to be obtained by summarizing the called routines. Because summarizing MOD and USE sets does not involve analysis of how data flows within routines, e.g. reaching-definition analysis, live variable analysis, array kill analysis, these summaries are referred to as flow-insensitive summaries.

The MOD and USE sets above are given for a statement. They can also be defined for an arbitrary code segment which has a single entry and a single exit. When these sets cannot be summarized exactly, an over-approximation, meaning that summary sets are super sets of the real sets, is computed so that dependence tests can still be conducted conservatively. Such an over-approximation set is also called a *may* set. To understand these summary schemes, we will briefly discuss the following two aspects: set representations and summaries of multiple array references over multiple control branches.

A. Set representations

Several set representations have been proposed so far, including convex regions [27, 28], data access descriptors [2] and bounded regular sections (for simplicity, we refer to this as regular sections when there is no confusion) [5, 15]. Although data access descriptors are more restrictive than convex regions, they basically define sets by a system of inequalities and equalities just as convex regions do. Data access descriptors also contain additional information besides the shapes of array accesses. Regular sections are simple and more restricted in representing power compared with convex regions and data access descriptors, but they still cover most cases in practice and can be implemented efficiently.

To show the difference between convex regions and regular sections, consider the following examples. The set of all array elements in a two dimensional array $A(N, M)$, where N and M are dimension sizes, can be represented exactly either by a convex region

$$< A(\psi_1, \psi_2), \{\psi_1 = I, \psi_2 = J, 1 \le I \le N, 1 \le J \le M\} >$$

or by a regular section

$$A(1 : N : 1, 1 : M : 1).$$

The upper triangular half of that array can also be represented exactly by a convex region:

$$< A(\psi_1, \psi_2), \{\psi_1 = I, \psi_2 = J, 1 \le I \le N, 1 \le J \le M, I \le J\} > .$$

However, this array region cannot be represented exactly by a regular section. It can only be approximated by $A(1 : N : 1, 1 : M : 1)$.

B. Summarizing references

To facilitate analysis, a code segment that will be summarized has a single entry point and a single exit point. In concept, this code segment can be viewed as a set of paths. Each path in this set is composed of a sequence of statements, from the entry point of the code segment to the exit point of the code segment, in the order of statement execution. The summary sets for each path are obtained by *composition* operations, which are union operations to combine all array references in the path together. The summary for the whole code segment is collected by merging the summary sets of all paths. Such merging operations, called *meet* operations, are union operations too. Also, all variables in the final summary sets should have their values on entry to the code segment. These summary schemes do not distinguish summary sets for different paths. Instead, they provide summary sets that are super sets of those of any path. We call these *path-insensitive* summaries.

In practice, a path-insensitive summary scheme can unionize all array references to form summary sets without considering paths, since both meet and composition operations are union operations. Variable replacements are carried out through the summarizing procedure in order to represent summary sets in terms of the input of the code segment. The typical code segments are loops, loop bodies, and routines.

Obviously, path-insensitive summary schemes cause *may* summary sets. The *may* sets can also be introduced when subscripts of array references are complicated (e.g. with expressions not representable in the compiler). In addition, attempting to represent the summary in a single convex region or a single regular section can also cause *may* sets because union operations are not closed. Some previous approaches such as [18] use a list of references as a set representation to avoid union operations and thus maintain exact summary information.

2.2 Array Data Flow Summaries

Recently, array data flow summaries, which analyze the intervening array kills, have been investigated [11, 25, 17, 29, 14, 7]. In these array data flow summaries, sets of upward exposed array uses (UE sets), which cannot be calculated without considering the effects of array kills, are calculated in addition to MOD and USE sets. The UE sets corresponding to a code segment contain all array elements whose values are imported to the code segment. The computation of UE sets usually needs difference operations. Let us look at an example. Suppose that statement S is in a given code segment and USE_s denotes all the array elements used by S. Let $MOD_{<s}$ be the set containing all array elements written before S within the code segment. These array summary sets are assumed to be corresponding to the same array A. To compute the UE sets for the code segment contributed by USE_s, we perform the following difference:

$$USE_s - MOD_{<s}$$

When $MOD_{<s}$ is a *may* set, the difference should be conservatively approximated by USE_s. Thus, $MOD_{<s}$ does not affect the computation of UE at all. Since this causes flow-sensitive analysis to degenerate, a *must* set (an under-approximation set in contrast to an over-approximation *may* set), which is definitely accessed, is introduced. Whenever the MOD set is inexact, we can use a *must* set in a difference operation. Therefore, the difference $USE_s - MOD_{<s}$ can be safely performed if $MOD_{<s}$ is a *must* set. Such a difference result is more precise even though it is still a *may* set.

The meet operations for *must* sets are intersection operations, while composition operations are still union operations. Must sets are calculated by intersecting summary sets for different branches at branching statements. The meet operations for *must* sets are closed. However, difference operations, which are introduced by computing UE sets, are not closed.

Previous experiments [3, 21] as well as our experience indicate that the path-insensitive MUST/MAY array data flow summary is not enough; IF conditions need to be considered in order to handle certain important cases in real programs. Take Figure 2.1 as an example. The MUST/MAY summary for the body of the outer DO will produce

$$may\ MOD : A(jlow : jup : 1), A(jmax : jmax : 1)$$

$$may\ UE : A(jmax : jmax : 1)$$

The intersection of these two sets is nonempty and hence the compiler has to conservatively assume that there exist loop-carried flow dependences. However, the IF condition $(.NOT.p)$ is loop-invariant, which guarantees that no loop-carried flow dependence exists in the outer loop, and therefore array A is privatizable. (We will discuss array privatization in section 6.)

In the next section, we propose a path-sensitive summary scheme which uses guarded array regions [12] to handle IF conditions. In order to preserve

```
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
```

Fig. 2.1. Examples of Privatizable Arrays

the exactness of union and difference results, we devise a scheme based on lists of guarded array regions. Such scheme turns out to enhance the efficiency of our approach as well.

3. Guarded Array Regions

Our analysis is based on two basic sets which describe array references, the upward exposed use set (UE set) and the modification set (MOD set). The UE set is the set of the upward exposed array elements which take values defined outside a given program segment. The MOD set is the set of array elements written within a given program segment.

Our basic unit of array reference representation is a *regular array region*, which is also called a bounded regular section [15]. It is a reduced form of the original regular sections proposed by Callahan and Kennedy [5]. On the other hand, we extend original regular sections in the following ways to meet our need to represent UE and MOD sets. First, since references to an array often cannot be easily represented by a single regular section without losing exactness, we use a list of regular array regions instead. In addition, we augment regular array regions by adding predicates as guards in order to better deal with IF conditions. The following gives the definitions of regular array regions and *guarded array regions.*

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. $(l : u : s)$ is simply denoted by (l) if $l = u$ and by $(l : u)$ if $s = 1$. An empty array region is represented by $\emptyset$ and an unknown array region is represented by Ω.

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 $\emptyset$.

The regular array region defined above is more restrictive than the original regular section used in the ParaScope environment at Rice University [5, 2, 15]. This basic unit, however, will be able to cover the most frequent cases in real programs and it seems to have an advantage in efficiency when dealing with the common cases. The guards in GAR's can be used to describe more complex array sections, although their primary use is to describe IF conditions under which regular array regions are accessed.

An *unknown* value is used carefully in the compiler in order to preserve as much precision as possible. First, one *unknown* dimension in a multiple dimensional array does not make the whole region *unknown*. Also, if the upper bound in a range tuple $(l : u : s)$ is unknown, we mark it as $(l : unknown : s)$ instead of as an unknown tuple.

We use a list of GAR's to represent a MOD set. However, using a list of GAR's for a UE set could introduce either more predicate operations [12] or unnecessary sacrifice of the exactness of difference operation results. Our solution to this problem is to define a data structure called a *GAR with a difference list (GARWD)*. A UE set is represented by a list of GARWD's.

Definition A *GAR with a difference list (GARWD)* is a set defined by two components: a *source* GAR and a difference list. The source GAR is an ordinary GAR defined above, while the difference list is a list of GAR's. The set denotes all the members of the source GAR which are not in any GAR of the difference list. It is written as { *source GAR*, <*difference list*> }.

DO I = 1, M	DO I = 1, M
A(1:N:1)=.	MOD_i: A(1:N:1), A(N2:N2:1)
A(N2) = .	
......	UE_i: { A(2:N1:1), < A(1:N:1), A(N2:N2:1) >}
. = A(2:N1:1)	
ENDDO	ENDDO

Fig. 3.1. Example of GARWD's

Figure 3.1 is an example showing the use of GARWD's. The right-hand side is the MOD/UE summary for the body of the outer loop shown on the left-hand side. The subscript i in both sets means the summary is for an arbitrary iteration i. UE_i is represented by a GARWD. For simplicity, we omit the guards here whose values are true in the example. To prove that no loop-carried data flow dependence exists, we need to show that the intersection of UE_i and the set of all mods before iteration i is empty. The set of all mods within iterations prior to iteration i, denoted as $MOD_{<i}$, is equal to MOD_i. (In theory, $MOD_{<i} = \phi$ if $i = 1$. But this does not invalidate the analysis above. Similarly, $MOD_{>i}$ denotes the sets of mods within the iterations after iteration i.) Since both GAR's in the $MOD_{<i}$ list are in the difference list of the UE_i, represented by a GARWD, it is obvious that the intersection of $MOD_{<i}$ and UE_i is empty. As will be discussed later in Section 6, this further shows that array A is privatizable.

3.1 Operations on GAR's

Our data flow analysis requires three kinds of operations on GAR's: 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. We will first describe the operations on array regions, then on GAR's, and finally on GARWD's.

Regular array region operations

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, $R_1 = A(r_1^1, r_2^1, \cdots, r_m^1)$ and $R_2 = A(r_1^2, r_2^2, \cdots, r_m^2)$, 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 in both R_1 and R_2. Other step values will be handled in Section 5. Let $r_i^1 = (l_i^1 : u_i^1 : 1)$, $r_i^2 = (l_i^2 : u_i^2 : 1)$, where $i = 1, \cdots, m$, and let D_i be $r_i^1 \cap r_i^2$. We have $D_i = (max(l_i^1, l_i^2) : min(u_i^1, u_i^2) : 1)$ and have $R_1 \cap R_2$ equal to

$$\begin{cases} \emptyset & \exists i, D_i = \emptyset \\ (D_1, D_2, \cdots, D_m) & Otherwise \end{cases}$$

  Note that we do not keep max and min operators in a regular array region. Therefore, when the relationship of symbolic expressions cannot be determined even after a demand-driven symbolic analysis is conducted (*c. f.* Section 5), we will mark the intersection as unknown.
- $R_1 \cup R_2$:
  Since regular array regions may contain symbolic terms, care must be taken to prevent invalid regions being created by union operations. For example, for $R_1 = (m : p : 1)$ and $R_2 = (p+1 : n : 1)$, we have $R_1 \cup R_2 = (m : n : 1)$ if and only if both R_1 and R_2 are valid. The validity of this union result can be guaranteed by inserting validity predicates into guards [12]. However, since this introduces additional predicate operations which we try to avoid, we represent the union as a list of regions without merging them until they are known to be valid, e.g. when they are both constant regions.

- $R_1 - R_2$:
  For an m-dimensional array, the result of the difference operation is generally 2^m regular regions if each range difference results in two new ranges. Such a result could be quite complex for large m. Nonetheless, it is useful to present a general formula for the result. 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. According to this definition, 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 $R_2(m-1) = (r_2^2, r_3^2, \cdots, r_m^2)$. The computation of $R_1 - R_2$ is recursively given by the following formula:

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

  To avoid a potentially complex result and also to keep the summary information precise, we use GARWD's to represent difference operations whenever the resultant difference is not a single regular region. We record the difference operation in the GARWD's without actually performing it until it is actually needed. As will be discussed in Section 5, the GARWD's also improve the efficiency of our overall scheme.

  The following gives examples of the actual results of difference operations,

- $(1:100) - (2:100) = (1)$
- $(1:100, 1:100) - (3:99, 2:100)$
  $= ((1:100) - (3:99), (1:100)) \cup ((3:99), ((1:100) - (2:100))$
  $= (((1:2) \cup (100)), (1:100)) \cup (3:99, 1)$
  $= (1:2, 1:100) \cup (100, 1:100) \cup (3:99, 1)$

GAR operations:

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

- $T_1 \cap T_2 = [P_1 \wedge P_2, R_1 \cap R_2]$
- $T_1 \cup T_2$
  The following are two most frequent cases of union operations:
  - If $P_1 = P_2$, the union becomes $[P_1, R_1 \cup R_2]$
  - If $R_1 = R_2$, the result is $[P_1 \vee P_2, R_1]$

  If two array regions cannot be safely combined due to the unknown symbolic terms, we will keep two GAR's in a list without merging them.
- $T_1 - T_2 = [P_1 \wedge P_2, R_1 - R_2] \cup [P_1 \wedge \overline{P_2}, R_1]$
  As described earlier, the actual result of $R_1 - R_2$ may be multiple regular array regions, which makes the actual result of $T_1 - T_2$ potentially complex. However, as Figure 3.1 illustrates, difference operations can often be canceled by intersection and union operations. Therefore, we do not solve the difference $T_1 - T_2$ unless the result is a single GAR or until the last moment when the actual result must be solved in order to finish a data

dependence test or an array privatizability test. When the difference is not solved with the above formula, it is represented by a GARWD.

The following shows examples of actual results of GAR operations:

- $[T, (1:100)] \cap [p, (2, 101)] = [p, (2:100)]$
- $[T, (1:50)] \cup [T, (51, 100)] = [T, (1:100)]$
  $[T, (1:100)] \cup [p, (1, 100)] = [T, (1:100)]$
- $[T, (1:100)] - [T, (2:99)]$
  $= [T, ((1:100) - (2:99))] = [T, ((1) \cup (100))]$
  $= [T, (1)] \cup [T, (100)]$

- $[p, (2:99)] - [T, (1:100)] = [p, ((2:99) - (1:100))] = [p, \emptyset] = \emptyset$

GARWD operations:

Operations between two GARWD's and between a GARWD and a GAR can be easily derived from the operations on GAR's. Some examples are given below to illustrate these operations:

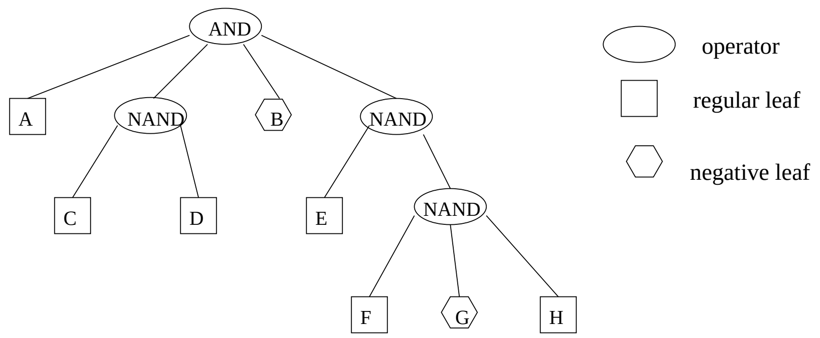
- $\{[T, (1:100)], < [T, (n:m)] >\} - [T, (2:100)]$
  $= \{([T, (1:100)] - [T, (2:100)]), < [T, (n:m)] >\}$
  $= \{[T, (1)], < [T, (n:m)] >\}$
- $\{[T, (1:100)], < [T, (n:m)] >\} \cap [p, (101:200)]$
  $= \{([T, (1:100)] \cap [p, (101:200)]), < [T, (n:m)] >\}$
  $= \{(\emptyset), < [T, (n:m)] >\} = \emptyset$
- $\{[T, (1:100)], < [T, (n:m)] >\} \cup \{[T, (1:100)], <>\} = \{[T, (1:100)], <>\}$

3.2 Predicate Operations

Predicate operations are expensive in general and most compilers avoid analyzing them. However, the majority of predicate-handling required for collecting our array data flow summary involves simple operations such as checking to see if two predicates are identical, if they are loop-independent, or if they contain indices that affect the shapes or the sizes of array regions. These relatively simple operations can be implemented in an efficient way.

We define a new canonical form to represent predicates in order to simplify pattern matching needed to check whether two predicates are identical. Both conjunctive normal form (CNF) and disjunctive normal form (DNF) have been widely used in program analysis [26, 6]. These cited works show that negation operations are expensive with both CNF and DNF. Our previous experiments using CNF [12] also confirm their observations. Negation operations occur not only due to ELSE branches, but also due to GAR and GARWD operations elsewhere. Hence, we design a new canonical form such that negation operations can be reduced significantly.

We use a two-level hierarchical approach to predicate handling. At the high level, a predicate is represented by a predicate tree, $PT(V, E, r)$, where V is the set of nodes, E is the set of edges, and r is the root of PT. The internal



$$A \cap \overline{(C \cap D)} \cap \overline{B} \cap \overline{(E \cap \overline{(F \cap \overline{G} \cap H)})}$$

Fig. 3.2. High Level Representation of Predicates

nodes of V are NAND operators except the root, which is an AND operator. The leaf nodes are divided into *regular leaf nodes* and *negative leaf nodes*. A regular leaf node represents a predicate such as an IF condition or a DO condition in the program, while a negative leaf node represents the negation of a predicate. Theoretically, the predicate tree may not be unique for an arbitrary predicate, which can cause pattern-matching to be conservative. We believe, however, that such cases are rare and happen mostly when the program cannot be parallelized.

Figure 3.2 shows a predicate tree. At this high level, we keep a basic predicate as a unit and do not split it. The predicate operations are based only on these units without further checking the contents within these basic predicates. Figure 3.3 show predicate operations. Figure 3.3(a) and (b) show OR and AND operations respectively. Negation of a predicate tree, shown in Figure 3.3(c), may either increase or decrease the tree height by one level according to the shape of the current predicate tree. In Figure 3.3(c), if there is only one regular leaf node (or one negative leaf node) in the tree, the regular leaf node is simply changed to a negative leaf node (or vice versa). We use a unique token for each basic predicate such that simple and common cases can be easily handled without checking the contents of the predicates. At the lower level of the hierarchy, the content of each predicate is represented in CNF and is examined when necessary.

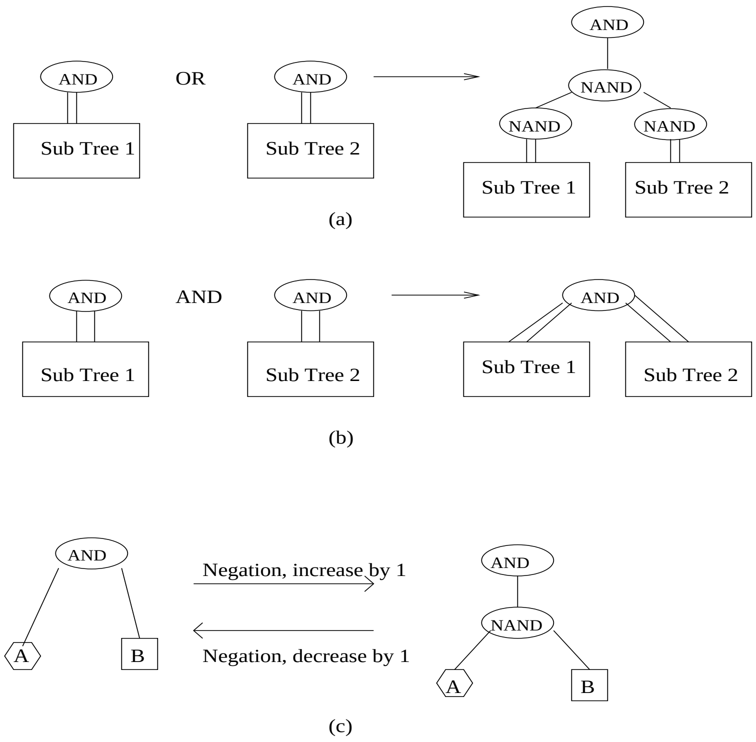


Fig. 3.3. Predicate Operations

4. Constructing Summary GAR's Interprocedurally

4.1 Hierarchical Supergraph

In this section, we present algorithms to calculate the MOD and UE information by propagating the GAR's and the GARWD's over a *hierarchical supergraph* (HSG). The HSG in this paper is an enhancement of Myers' *supergraph* [22] which is a composite 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, *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 internal flow subgraphs

```
... ....
IF (X > 10) THEN
   B =  10
   CALL sub1(A,B)
ELSE
   B = 20
   CALL sub1(A,B)
ENDIF
C=...
... ....
SUBROUTINE sub1(A,M)
 DO I=1, m
    A(I) = ..
 ENDDO
END
```

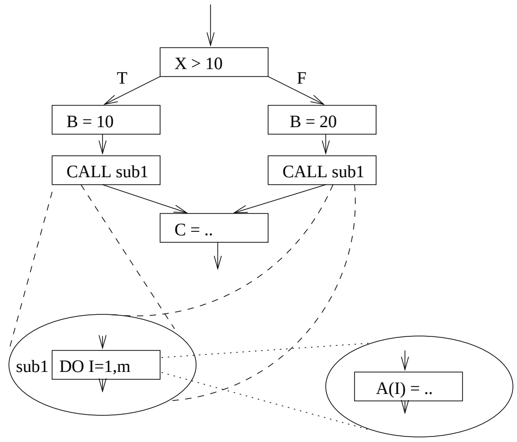


Fig. 4.1. Example of the HSG

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 [16]. Figure 4.1 shows a HSG example. 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 premature exits in DO loops and backward GOTO statements, making conservative estimates when necessary. 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.

4.2 Summary Algorithms

Suppose we want to check whether loop L is parallel. Let $g(s, e)$ be the HSG of the subgraph of loop L (the graph for the loop body), where s is the starting node and e the exit node. We use $UE(n)$ and $MOD(n)$ to denote the upward exposed use set and the mod set for node n respectively. Furthermore, we use $MOD\_IN(n)$ to represent the array elements modified in nodes which are reachable from n, and we use $UE\_IN(n)$ to represent the array elements whose values are imported to n and used in the nodes reachable from n.

Each $MOD(n)$ or $MOD\_IN(n)$ set is represented by a list of GAR's and each $UE(n)$ or $UE\_IN(n)$ set by a list of GARWD's. If any GAR contains variables in its guard or its range triples, these variables should assume the values imported at the entry of n. The compiler performs the following recursive algorithm, *gar_summary*:

1. Summarize all basic block nodes within $g(s, e)$.
   For each basic block node n, we calculate the $MOD(n)$ and $UE(n)$ sets for arrays within node n. All predicates in GAR's of $MOD(n)$ and $UE(n)$ are set to True.
2. Summarize all compound nodes.
   a) *Call nodes.* For a call node n, let the subgraph of the call node be $g'(s, e)$. The compiler recursively applies *gar_summary* to $g'(s, e)$, which summarizes the references to global arrays and arrays parameters. The returning results are mapped back to the actual arguments of the procedure call and are denoted by $MOD(n)$ and $UE(n)$.
   b) *loop nodes.* For a loop node n, let the subgraph of the loop node be $g'(s, e)$. The compiler recursively applies *gar_summary* to $g'(s, e)$. Let $UE_i(n)$ and $MOD_i(n)$ represent the UE and MOD sets of $g'(s, e)$, which contain modification GAR's and upwards exposed use GARWD's in one loop iteration indexed by i. The sets $MOD_i(n)$ and ($UE_i(n)$ - $MOD_{<i}(n)$) are then expanded across the i index range to form $MOD(n)$ and $UE(n)$ for loop node n
3. Propagate the array data flow information.
   From node e to s, the *gar_summary* algorithm traverses the nodes in $g(s, e)$ in a reverse topological order. During the summary propagation, the following flow equations are applied:
   $$MOD\_IN(n) = MOD(n) \cup (\textstyle\bigcup_{p \in succ(n)} MOD\_IN(p))$$
   $$UE\_IN(n) = UE(n) \cup (\textstyle\bigcup_{p \in succ(n)} UE\_IN(p) - MOD(n))$$
   Note that the set of successors of the exit node succ(e) is $\emptyset$. When applying the flow equations, we need to handle the following situations.
   – If n is a basic block containing an IF-condition, insert the corresponding predicate or its negation into the guard of each GAR which has been propagated backward into $MOD\_IN(n)$ and $UE\_IN(n)$ through the THEN edge or the ELSE edge respectively.
   – If any expression in the $MOD\_IN(p)$ and $UE\_IN(p)$ 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. Therefore, all values of variables in $MOD\_IN(n)$ and $UE\_IN(n)$ are relative to the entry to n.

For loop L, the above algorithm produces the MOD_i and UE_i sets of $g(s, e)$, where MOD_i equals $MOD\_IN(s)$ and UE_i equals $UE\_IN(s)$. These sets can be used to do dependence tests and array privatization tests, which are presented later in this chapter.

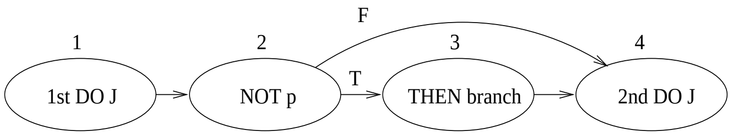


Fig. 4.2. The HSG of the Body of the Outer Loop for Figure 2.1

Let us use the example in Figure 2.1 to illustrate the algorithm. A simplified HSG of the body of the outer loop is shown in Figure 4.2, which omits the internal flow graphs of the compound nodes. Following the algorithm above, Step 1 and Step 2 give us the following results:

$$
\begin{array}{lll}
MOD(1) = & [T,(jlow:jup)], & UE(1) = \emptyset \\
MOD(2) = & \emptyset, & UE(2) = \emptyset \\
MOD(3) = & [T,(jmax)], & UE(3) = \emptyset \\
MOD(4) = & \emptyset, & UE(4) = [T,(jlow:jup)] \cup [T,(jmax)]
\end{array}
$$

Step 3, following a reverse topological order, propagates the information in the following four sub-steps:

1. $MOD\_IN(4) = MOD(4) = \emptyset$
   $UE\_IN(4) = UE(4) = [T,(jlow:jup)] \cup [T,(jmax)]$
2. $MOD\_IN(3) = MOD(3) \cup MOD(4) = [T,(jmax)]$
   $UE\_IN(3) = UE(3) \cup (UE\_IN(4) - MOD(3)) = \{[T,(jlow:jup)], < [T,(jmax)] >\}$
3. $MOD\_IN(2) = [\overline{p},(jmax)]$
   $UE\_IN(2) = \{[\overline{p},(jlow:jup)], < [\overline{p},(jmax)] >\} \cup [p,(jlow:jup)] \cup [p,(jmax)]$

   In the above, $\overline{p}$ is inserted into the guards of the GAR's which are propagated through the TRUE edge and p is inserted into those propagated through the FALSE edge.
4. $MOD\_IN(1) = [T,(jlow:jup)] \cup [\overline{p},(jmax)]$
   $UE\_IN(1) = UE\_IN(2) - MOD(1) = \{[p,(jmax)], < [T,(jlow:jup)] > \}$

Therefore, the summary sets of the body of the outer loop (DO I) should be:

$$
\begin{aligned}
MOD_i &= MOD\_IN(1) = [T,(jlow:jup)] \cup [\overline{p},(jmax)] \\
UE_i &= UE\_IN(1) = \{[p,(jmax)], < [T,(jlow:jup)] >\}
\end{aligned}
$$

4.3 Expansions

Given a loop with index i, where $i \in (l:u:s)$, and MOD_i and UE_i as the summary sets for the loop body, what are the UE and MOD sets for the loop? This problem can be solved by an *expansion* procedure which is conducted in the following two steps:

$$UE(i) = UE_i - MOD_{<i}, \ where \ \ MOD_{<i} = \Sigma_{j \in (l:u:s) \wedge (j<i)} MOD_j, \qquad (I)$$

$$\begin{cases} MOD &= \cup_{i \in (l:u:s)} MOD_i \\ UE &= \cup_{i \in (l:u:s)} UE(i) \end{cases} \qquad (II)$$

Step I in the above computes the set of array elements used in iteration i which are exposed to the outside of the *whole loop*. Step II projects the result of Step I to eliminate i. The projection for a list of GAR's is conducted by the projection for each GAR separately. Suppose Q is a GAR. If Q does not contain i in its representation, then the projection of Q is Q itself. If Q contains i in its representation, then the projection of Q by i, denoted by proj(Q), is a GAR obtained by the following steps:

- If i appears in the guard of Q, then i should be solved from the guard which, in general, is written as $i \in (l' : u' : s)$, where l' and u' may be symbolic expressions. We obtain a new domain of i as follows,

  $$(\lceil \frac{max(l', l) - l}{s} \rceil \cdot s + l : \lfloor \frac{min(u', u) - l}{s} \rfloor \cdot s + l : s)$$

  which simplifies to $(max(l', l) : min(u', u))$ for $s = 1$. The inequalities and equalities involving i in the guard can then be deleted.
  For example, given $i \in (2 : 100 : 2)$ and GAR $[5 \le i, A(i)]$, the new domain of i should be $(6 : 100 : 2)$. $5 \le i$ is removed from the GAR.
- If i appears in only one dimension of Q, and if the result of substituting $l \le i \le u$, or the new bounds on i obtained above, into the old range triple in that dimension can still be represented by a range triple $(l'' : u'' : s'')$, then we replace the old range triple by $(l'' : u'' : s'')$.
- If, in the above, the result of substitution of $l \le i \le u$ into the old range can no longer be represented by a range, then we mark this range as Ω (unknown). (Better approximation is possible for special cases, but we have not pursued further.)
- If i appears in more than one dimension of Q, then these dimensions are marked as unknown.

As an example, suppose a DO loop indexed by i has loop bounds $1 \le i \le 100$. Further, suppose the given GAR is $Q = [3 \le i + 1 \le 51, (1 : i : 1)]$. As the result of solving i from the guard, we have new bounds on i which are $(max(1, 3 - 1) : min(100, 51 - 1)) = (2 : 50)$. The expansion of Q by i is $[True, 1 : 50 : 1)]$.

Since the UE_i set is represented by a list of GARWD's, the expansion of UE_i can be obtained by unionizing the expansion of the individual GARWD's in UE_i. Therefore, for each GARWD W in UE_i, we need to compute $(W - MOD_{<i})$ and then its projection. If i cannot be removed from the difference

list of a GARWD by simplifying the GARWD, we over-approximate this GARWD by removing the whole difference list from the GARWD. As we mentioned previously, GARWD is used to represent UE sets only. Therefore, over-approximation is always safe. Afterwards, each GARWD in the UE_i set either has an empty difference list or one that is loop-invariant. Now only the source GAR of a GARWD needs to be considered below.

The difference $(W - MOD_{<i})$ is calculated by successive subtractions of each GAR of $MOD_{<i}$ from W. Let Q_i be the source GAR of $W, i \in (l : u : s)$. The details of the expansion of a single GARWD W, including Step I and Step II, are as follows:

(**A**) If Q_i is loop-invariant, then we have the expansion of W equal to $proj(W - MOD_{<i}) = W$. Stop.
(**B**) For (each M in $MOD_{<i}$) {

1. If M is loop-invariant and Q_l (the value of Q for the first iteration) does not overlap with M, we insert M into the difference list of W. Goto *continue*.
2. If M is loop-invariant and Q_l is equal to M, then M does not affect the expansion. Goto *continue*.
3. If $Q_i \cap M = \phi$, then $Q_i - M = Q_i$. Goto *continue*.
4. Suppose both Q_i and M are loop-variant and the index i appears in the same dimension, say dimension j, in both Q_i and M. Further suppose that Q_i and M are identical in all dimensions except dimension j, in which the range tuples contain one single element and can be represented by (E_t) and(E_m) respectively. If $(E_m - E_t)$ is not a constant, goto step 5. If $(E_m - E_t)$ is a constant, the coefficients of i in both Q_i and M must be the same. Let this coefficient be coe_i. (If $(E_m - E_t)$ is not divisible by $(coe_i \cdot s)$, then $Q_i \cap M = \phi$ and should be covered in step 3.) When both coe_i and s are greater than zero, the Q_i in the first $(E_m - E_t)/coe_i$ number of iterations is exposed to the outside of the loop. Therefore,

$$(Q_i - M), i \in (l : u : s) =$$
$$\begin{cases} Q_i,\ i \in (l : l + (E_m - E_t)/coe_i - s : s); & (E_m - E_t) > 0 \\ Q_i,\ i \in (l : u : s); & (E_m - E_t) \le 0 \end{cases}$$

   For the first case, a new domain for i is established. l and u will have new values for the rest of the computation on $(W - MOD_{<i})$.
   For other cases in which coe_i or s is not greater than zero, the formula can be acquired similarly.
   Goto *continue*.
5. We mark W as a may set and drop the difference list of W. Compute proj(W), which is the final result of the expansion of the original W. Stop.
6. *continue:*

}
(**C**) Compute proj(W), which is the final result of the expansion of the original W.

The above steps have been implemented currently and can be extended and refined. The projection of a GARWD W, denoted above by proj(W), can be acquired by projections on each GAR in W separately.

5. Implementation Considerations

5.1 Symbolic Analysis

Symbolic analysis handles expressions which involve unknown symbolic terms. It is widely used in symbolic evaluation and abstract interpretation techniques to discover program properties such as values of expressions, relationships between symbolic expressions, etc. Symbolic analysis requires to represent and manipulate unknown symbolic terms. A symbolic analyzer can adopt a certain normalized form to represent expressions [26, 6, 13]. The advantage of such a normalized form is that it gives the same representation for two *congruent expressions*, i.e. they always have the same value. For example, $x \cdot (x+1)$ and $x + x^2$ are congruent expressions. In addition, symbolic expressions encountered in array data flow analysis and dependence analysis are mostly integer polynomials. Many operations on integer polynomials, such as the comparison of two polynomials, can be straightforward if a normalized form is adopted. Therefore, we adopt normalized integer polynomials as our representation for symbolic expressions, whose form is shown below:

$$e = \Sigma_{i=1}^{N} t_i \cdot I_i + t_0 \tag{5.1}$$

where each I_i is an index variable and t_i is a term which is given by equation (2) below:

$$t_i = \Sigma_{j=1}^{M_i} p_j, \quad i = 1, \cdots, N \tag{5.2}$$

$$p_j = c^j \cdot \Pi_{k=1}^{L_j} x_k^j, \quad j = 1, \cdots, M_j \tag{5.3}$$

where p_j is a product, c^j is an integer constant (possibly an integer fraction), x_k^j is an integer variable which is not an index variable, N is the nesting depth of the loop containing e, M_i is the number of products in t_i, and L_j is the number of variables in p_j. All e, t_i, and p_j are sorted by each variable's order number, which is a unique integer number assigned to each variable. Since both M_i and L_j contribute to the complexity of a polynomial, they are chosen as design parameters which can be adjusted in the compiler, according to experimental results, in order to control the complexity of expressions. As an example, by limiting M_i to be 1 and L_j to be zero, the expression e will become an affine expression. By controlling the complexity of expressions, the complexity of the overall summary scheme is reduced.

We implement utility library functions to perform symbolic operations such as additions, subtractions, multiplications, and divisions by an integer constant. In addition, a simple demand-driven symbolic evaluation scheme is implemented. It propagates an expression backwards over a control flow graph until either the value of the expression becomes known or a predefined propagation limit is reached.

5.2 Region Numbering

During the propagation and manipulation of regular array regions, GAR's, and GARWD's, each of these data structures may be duplicated several times. Operations involving these identical copies can potentially be time-consuming if straightforward pattern-matching is conducted. Instead, we introduce a region numbering scheme. In this scheme, the same number is assigned to the identical copies of an array region. Figure 5.1 shows an example. Initially,

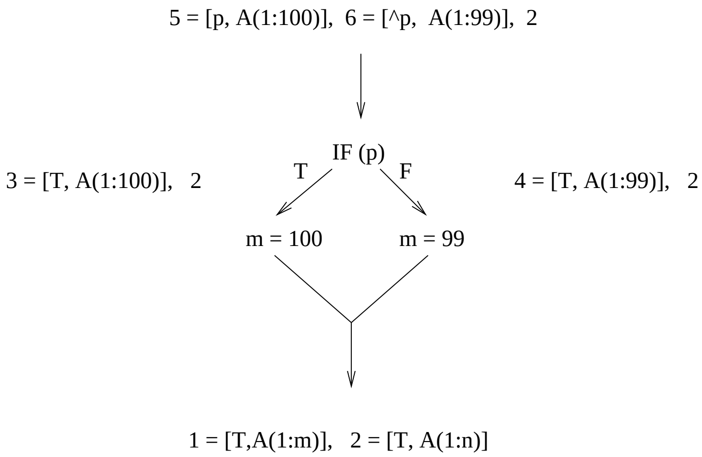


Fig. 5.1. Propagation Using Region Numbers

there are two GAR's, numbered 1 and 2. As GAR No.1 is propagated through both branches, it is modified and becomes GAR No.3 in the left branch and No.4 in the right branch, while GAR No.2 remains unchanged. At the entrance to the IF, GAR No.3 is modified by adding predicate p and becomes GAR No.5, while GAR No.4 becomes GAR No.6. Since GAR No.2 appears in both branches (the number 2 appears in both lists), the IF condition is not attached to it. Thus GAR No.2 is unchanged.

However, we do not try to use the same number to represent identical array regions which originate from different array references in the program, because that may require us to compare many array regions from time to time. Instead, we use a simplifier to remove redundant array regions. The simplifier is invoked at the end of a routine summarizing or immediately before the data dependence tests and array privatization tests.

Predicates are handled similarly. We assign each leaf node and NAND node a unique number to facilitate the operations on predicates.

5.3 Range Operations

In this subsection, we give details of range operations for step values of other than one. These operations invoke the $min(e_1, e_2)$ and the $max(e_1, e_2)$ functions. Each function either returns a symbolic expression containing neither min nor max function calls, or it returns an "unknown" value. Symbolic analysis is performed on demand.
Given two ranges r_1 and r_2, $r_1 = (l_1 : u_1 : s_1)$, $r_2 = (l_2 : u_2 : s_2)$, the following gives the results of range operations:

1. If $s_1 = s_2 = 1$,
   - $r_1 \cap r_2 =$
     $(max(l_1, l_2) : min(u_1, u_2) : s_1)$
   - Assuming $r_2 \subseteq r_1$ (otherwise use $r_1 - r_2 = r_1 - r_1 \cap r_2$), we have
     $r_1 - r_2 = (l_1 : max(l_1, l_2) - 1 : s_1) \cup (min(u_1, u_2) + 1 : u_1 : s_1)$
   - 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, $r_1 \cup r_2 = (min(l_1, l_2) : max(u_1, u_2) : s_1)$, assuming r_1 and r_2 are both valid.

   The result is returned to the upper level routine, in which whether to perform these operations, keep them in a list, etc., is decided.
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 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 cannot be combined into one range and must be maintained as a list of ranges.
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 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 $r_1 \cup r_2 = r_2$. Otherwise, we divide r_2 into several smaller ranges with step s_1 and then apply the above formulas.
5. In all other cases, the results of intersection and difference are marked as unknown and the union remains a list of two ranges.

6. Application to Array Privatization and Preliminary Experimental Results

In this section, we first discuss how our array data flow summary can be used to perform array privatization even in difficult cases which are sensitive to IF conditions. We then present preliminary experimental results to show both the capability and the efficiency of our analysis.

6.1 Array Privatization

Array privatization has been shown to be one of the most important techniques to improve the performance of parallelizing compilers [3, 4, 12, 21]. This technique creates a distinct copy of an array for each processor such that storage conflicts can be eliminated without violating program semantics. Privatizable arrays appear in real programs quite often and usually serve as temporary working spaces. It has been shown that traditional data dependence tests cannot identify privatizable arrays. In contrast, data flow analysis, when supported by IF condition handling, symbolic analysis, and interprocedural analysis, allows a powerful compiler to recognize privatizable arrays.

An array A is a privatization candidate in a loop L if its elements are overwritten in different iterations of L (see [17]). Such a candidacy can be established by examining the summary array MOD_i set: If the intersection of MOD_i and $MOD_{<i}$ is nonempty, then A is a candidate. 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.

Let us look at Figure 2.1 again. In Figure 4.2, we have computed MOD_i and UE_i. Since MOD_i is not loop-variant, we have $MOD_{<i} = MOD_i$. Hence, $MOD_i \cap MOD_{<i}$ is nonempty and array A is a privatization candidate. Furthermore,

$$
\begin{aligned}
&UE_i \cap MOD_{<i} \\
&\quad = \{[p,(jmax)], < [T,(jlow:jup)] >\} \cap ([T,(jlow:jup)] \cup [\overline{p},(jmax)]) \\
&\quad = \{[p,(jmax)], < [T,(jlow:jup)] >\} \cap [T,(jlow:jup)] \cup \\
&\quad \{[p,(jmax)], < [T,(jlow:jup)] >\} \cap [\overline{p},(jmax)] \\
&\quad = \{[p,(jmax)], < [T,(jlow:jup)] >\} \cap [T,(jlow:jup)] \cup \\
&\quad \{([p,(jmax)] \cap [\overline{p},(jamx)]), < [T,(jlow:jup)] >\} \\
&\quad = \{[p,(jmax)], < [T,(jlow:jup)] >\} \cap [T,(jlow:jup)] \\
&\quad = \emptyset
\end{aligned}
$$

The last difference operation above can be easily done because GAR $[T,(jlow: jup)]$ is in the difference list. The fact of $UE_i \cap MOD_{<i}$ being empty guarantees that array A is privatizable.

6.2 Preliminary Experimental Results

We have implemented our array data flow analysis in a prototyping parallelizing compiler, Panorama, which is a multiple-pass, source-to-source, Fortran program restructurer [23]. It performs parsing, construction of a hierarchical supergraph(HSG) and interprocedural scalar UD/DU chains [1], conventional data dependence tests, array data flow analysis and other analyses which support memory performance enhancement, and parallel code generation.

Table 6.1 shows several time-consuming Fortran loops in the Perfect benchmark suite which are parallelized after our array data flow analysis and array privatization. (Additional transformations such as induction variable substitution are automatically performed to enable parallelization, whose discussions, however, are omitted in this paper.) The last column of Table 6.1 presents the current status of array privatization achieved by Panorama. All arrays listed in Table 6.1 can be privatized under our scheme except array RL in the MDG program. This case needs to handle predicates involving subscript variables, which our current implementation is unable to deal with [12].

Table 6.1. 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.

Table 6.2 shows the cpu time spent on the main parts of Panorama. In Table 6.2, parsing time is the time to parse the program once, although Panorama currently parses a program three times (the first time for constructing the call graph and for rearranging the parsing order of the source files, the second time for interprocedural analysis and the last time for code generation). We do not count the time spent on memory performance analyses in Table 6.2.

The row “HSG & DOALL Checking” shows the time taken to build the HSG and UD/DU chains and to do conventional DOALL checking. The “Array Summary” refers to our array data flow analysis, which is applied only to loops whose parallelizability cannot be determined by the conventional

Table 6.2. Execution Time (in seconds) Distribution (Timing is acquired on SGI Indy workstations with 134MHz MIPS R4600 CPU and 64 MB memory)

Program	TRACK	MDG	TRFD	OCEAN	ARC2D
#Lines	3784	1238	485	4343	3964
Parsing	1.78	0.86	0.40	1.26	2.43
HSG & DOALL Checking	3.07	1.72	0.49	0.91	5.16
Array Summary	6.01	2.38	0.50	0.36	27.72
Code Generation	1.62	0.82	0.22	0.42	2.26
Total	12.48	5.78	1.61	2.96	37.56
f77 -O	17.54	12.37	7.17	39.32	37.67

DOALL tests. Even though the time percentage of array data flow analysis is high (about 40%), the total execution time is small (38 seconds maximum). As an interesting comparison, the next row marked by “f77 -O” shows the time spent by the f77 compiler with option -O to compile the corresponding FORTRAN program into sequential machine code. This row is used as a reference point to better understand the speed of our parallelizing compiler. The data suggest that Panorama is quite efficient.

7. Related Works

There exist a number of approaches to array data flow analysis. As far as we know, no work has particularly addressed the efficiency issue or has presented efficiency data. One class of approaches attempt to gather flow information for each individual array reference. Feautrier [10] calculates the *source function* which represents the precise array definition points for each array use. Maydan, et al. [20, 21] simplify Feautrier's method by computing a Last-Write-Tree (LWT). Duesterwald, et al., compute the dependence distance for each reaching definition within a loop [8]. Pugh and Wonnacott [24] use a set of constraints to describe array data flow problems and solve them by an extented Fourier-Motzkin variable elimination. Maslov [19], as well as Pugh and Wonnacott [24], extends the previous work in this category by handling certain IF conditions. Generally, these approaches are intraprocedural and do not seem easy to extend interprocedurally.

The other class of approaches summarizes the array elements accessed by a set of array references instead of analyzing individual array references. This class extends data structures designed by the early works for flow-insensitive array problems, such as *regular array sections* [5, 15], convex regions [27, 28], and data access descriptors [2]. Array data flow analysis in this class includes works done by Gross and Steenkiste [11], Rosene [25], Li [17], Tu and Padua [29], Creusillet and Irigoin [7], and M. Hall, et al. [14]). The work described in this chapter seems to be the only one which considers the effect of IF conditions in a general way, although some previous works also handle IF conditions under more limited circumstances. Although this class of approaches does not provide as many details about reaching-definitions as the first one, it handles complex program constructs better and is easier to perform interprocedurally.

8. Conclusion

Array data flow information is important to successful automatic program parallelization. Propagating array data flow summary interprocedurally is an effective way to deal with procedure calls in program parallelization. Most existing forms of array data flow summaries are path-insensitive, i.e., they do not distinguish summary sets for different program paths. In certain important cases, such path-insensitivity hurts the result of parallelization badly.

In this chapter, we have proposed a path-sensitive array data flow summary, and we have described an efficient scheme to compute such a summary and use it for array privatization and program parallelization. Efficient symbolic processing is integrated into the process of propagating array access information. Preliminary results of array privatization suggest that our analysis is both powerful and efficient.

Acknowledgement. We thank Trung N. Nguyen and Guohua Jin for their important contributions to Panorama. 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 a National Science Foundation CAREER Award, Grant CCR-950254 and a funding, as a part of DICE project, from Samsung Electronics.

References

1. Aho, A.V., Sethi, R., Ullman, J.D. (1986). Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, Mass.
2. Balasundaram, V. (1990). A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. Journal of Parallel and Distributed Computing, 9:154–170.
3. Blume, W., Eigenmann, R. (1994). Symbolic Analysis Techniques Needed for the Effective Parallelization of Perfect Benchmarks. Technical report, Dept. of Computer Science, University of Illinois.
4. Blume, W., Eigenman, R. (1992). Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. IEEE Trans. on Parallel and Distributed Systems, November, 3(6):643-656.

5. Callahan, D., Kennedy, K. (1986). Analysis of interprocedural side effects in a parallel programming environment. ACM SIGPLAN '86 Symp. Compiler Construction, June, 162–175.
6. Clarke, L.A., Richardson, D.J. (1985). Applications of Symbolic Evaluation. The Journal of Systems and Software, 5(1):15-35
7. Creusillet, B., Irigoin, F. (1995). Interprocedural Array Region Analyses. Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing, held in Columbus, Ohio, August, No. 1033, In Lecture Notes in Computer Science, Springer-Verlag, Berlin, 46-60.
8. Duesterwald, E., Gupta, R., Soffa, M.L. (1993). A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations. ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation, June, 68-77.
9. Eigenmann, R., Hoeflinger, J., Li, Z., Padua, D. (1992). Experience in the Automatic Parallelization of Four Perfect Benchmark Programs. Lecture Notes in Computer Science, 589, Springer-Verlag.
10. Feautrier, P. (1991). Dataflow Analysis of Array and Scalar References. International Journal of Parallel Programming, February, 2(1):23-53.
11. Gross, T., Steenkiste, P. (1990). Structured dataflow analysis for arrays and its use in an optimizing compiler. Software – Practice and Experience, February, 20(2):133-155.
12. Gu, J., Li, Z., Lee, G. (1995). Symbolic Array Dataflow Analysis for Array Privatization and Program Parallelization. Supercomputing'95, December, San Diego, CA.
13. Haghighat, M.R., Polychronopoulos, C.D. (1994). Symbolic analysis for Parallelizing Compilers. Technical report, CSRD Report No. 1355, University of Illinois.
14. Hall,M, Murphy, B., Amarasinghe, S., Liao, S., Lam, M. (1995). Interprocedural Analysis for Parallelization. Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing, held in Columbus, Ohio, August, No. 1033, In Lecture Notes in Computer Science, Springer-Verlag, Berlin, 61-80.
15. Havlak, P., Kennedy, K. (1991). An implementation of interprocedural bounded regular section analysis. IEEE Trans. on Parallel and Distributed Systems, 2(3):350-360.
16. Irigoin, F., Jouvelot, P., Triolet, R. (1991). Semantical Interprocedural Parallelization: An overview of the PIPS Project. International Conference on Supercomputing, 244-251.
17. Li, Z. (1992). Array Privatization for Parallel Execution of Loops. ACM Int. Conf. on Supercomputing, July, 313-322.
18. Li, Z., Yew, P.C. (1988). Efficient interprocedural analysis for program parallelization and restructuring. ACM SIGPLAN Conf. on Parallel Programming: Experience with Applications, Languages and Systems, September, 85-99.
19. Maslov, V. (1994). Lazy Array Data-Flow Dependence Analysis. Proceedings of Annual ACM Symposium on Principles of Programming Languages, January, Portland, Oregon, 331-325.
20. Maydan, D.E., Amarasinghe, S.P., Lam, M.S. (1993). Array Data-Flow Analysis and its Use in Array Privatization. Proc. of the 20th ACM Symp. on Principles of Programming Languages, January, 2-15.
21. Maydan, D.E. (1992). Accurate Analysis of Array References. PhD thesis, October, Stanford University.
22. Myers, E. W. (1981). A Precise Interprocedural Data-Flow Algorithm. Proceedings of 8th Annual ACM Symposium on Principles of Programming Languages. January, 219-230.

23. Nguyen, T., Gu, J., Li, Z. (1995). An Interprocedural Parallelizing Compiler and its support for Memory Hierarchy Research. Lecture Notes in Computer Science 1033: 8th International Workshop on Languages and Compilers for Parallel Computing, Columbus, Ohio, August, 96-110.
24. Pugh, W., Wonnacott, D. (1993). An Exact Method for Analysis of Value-based Array Data Dependences. Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, Portland, OR, August.
25. Rosene, C. (1990). Incremental Dependence Analysis. Technical report, CRPC-TR90044, PhD thesis, Computer Science Department, Rice University.
26. Cheatham, T.E. Jr, Holloway, G.H., Townley, J.A. (1979). Symbolic Evaluation and the Analysis of Programs. IEEE Trans.on Software Engineering, July, 5(4):402-417.
27. Triolet, R., Irigoin, F., Feautrier, P. (1986). Direct Parallelization of CALL Statments. ACM SIGPLAN'86 Sym. on Compiler Construction, Palo Alto, CA, July, 176-185.
28. Triolet, R. (1985). Interprocedural Analysis for Program Restructuring with Parafrase. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.
29. Tu, P., Padua, D. (1993). Automatic Array Privatization. Proceedings of Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR, August, 500-521.
30. Wolfe, M. (1996). High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company.