Efficient Interprocedural Array Data-flow Analysis for Automatic Program Parallelization *

Junjie Gu † Zhiyuan Li ‡

Abstract

Since sequential languages such as Fortran and C are more machine-independent than current parallel languages, it is highly desirable to develop powerful parallelization tools which can generate parallel codes, automatically or semi-automatically, targeting different parallel architectures. Array data-flow analysis is known to be crucial to the success of automatic parallelization. Such an analysis should be performed interprocedurally and symbolically, and it often needs to handle the predicates represented by IF conditions. Unfortunately, such a powerful program analysis can be extremely time-consuming if not carefully designed. How to enhance the efficiency of this analysis to a practical level remains an issue largely untouched to date. This paper presents techniques for efficient interprocedural array data-flow analysis and documents experimental results of its implementation in a research parallelizing compiler. Our techniques are based on *guarded array regions* and the resulting tool runs faster, by one or two orders of magnitude, than other similarly powerful tools.

Key words: Parallelizing compiler, array data-flow analysis, interprocedural analysis, symbolic analysis.

Corresponding Author:

Professor Zhiyuan Li
Department of Computer Sciences,
Purdue University,
West Lafayette, IN 47907
Phone: (765) 494-7822
Fax: (765) 494-0739
Email: li@cs.purdue.edu

Accepted to IEEE Transactions on Software Engineering, the special issue on Architecture-Independent Languages and Software Tools for Parallel Processing

*This paper is based in part on the work previously presented in Proceedings of Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 157 – 167. Las Vegas, Nevada, June 18-21, 1997.

†Sun Microsystems, Inc., 901 San Antonio Road, UMPK16-303 Palo Alto, CA 94303. e-mail: jgu@eng.sun.com.

‡Department of Computer Sciences, Purdue University, West Lafayette, IN 47907. e-mail: li@cs.purdue.edu. Supported in part by NSF CCR-950254, MIP-9610379 and by Purdue Research Foundation.

1 Introduction

Program execution speed has always been a fundamental concern for computation-intensive applications. To exceed the execution speed provided by the state-of-the-art uniprocessor machines, programs need to take advantage of parallel computers. Over the past several decades, much effort has been invested in efficient use of parallel architectures. In order to exploit parallelism inherent in computational solutions, progress has been made in areas of parallel languages, parallel libraries and parallelizing compilers. This paper addresses the issue of automatic parallelization of practical programs, particularly those written in imperative languages such as Fortran and C.

Compared to current parallel languages, sequential languages such as Fortran 77 and C are more machine-independent. Hence, it is highly desirable to develop powerful automatic parallelization tools which can generate parallel codes targeting different parallel architectures. It remains to be seen how far automatic parallelization can go. Nevertheless, much progress has been made recently in the understanding of its future directions. One important finding by many is the critical role of array data-flow analysis [10, 17, 20, 32, 33, 37, 38, 42]. This aggressive program analysis not only can support *array privatization* [29, 33, 43] which removes spurious data dependences thereby to enable loop parallelization, but it can also support compiler techniques for memory performance enhancement and efficient message-passing deployment.

Few existing tools, however, are capable of *interprocedural* array data-flow analysis. Furthermore, no previous studies have paid much attention to the issue of the efficiency of such analysis. Quite understandably, rapid prototyping tools, such as SUIF [23] and Polaris [4], do not emphasize compilation efficiency and they tend to run slowly. On the other hand, we also believe it to be important to demonstrate that aggressive interprocedural analysis can be performed efficiently. Such efficiency is important for development of large-sized programs, especially when intensive program modification, recompilation and retesting are conducted. Taking an hour or longer to compile a program, for example, would be highly undesirable for such programming tasks.

In this paper, we present techniques used in the Panorama parallelizing compiler [35] to enhance the efficiency of interprocedural array data-flow analysis without compromising its capabilities in practice. We focus on the kind of array data-flow analysis useful for array privatization and loop parallelization. These are important transformations which can benefit program performance on various parallel machines. We make the following key contributions in this paper:

- We present a general framework to summarize and to propagate array regions and their access conditions, which enables *array privatization* and loop parallelization for Fortran-like programs which contain nonrecursive calls, symbolic expressions in array subscripts and loop bounds, and IF conditions that may directly affect array privatizability and loop parallelizability.

- We show a hierarchical approach to predicate handling, which reduces the time complexity of analyzing the predicates which control different execution paths.
- We present experimental results to show that reducing unnecessary set-difference operations contributes significantly to the speed of the array data-flow analysis.
- We measure the analysis speed of Panorama when applied to application programs in the Perfect benchmark suite [3], a suite that is well known to be difficult to parallelize automatically. As a way to show the quality of the parallelized code, we also report the speedups of the programs parallelized by Panorama and executed on an SGI Challenge multiprocessor. The results show that Panorama runs faster, by one or two orders of magnitude, than other known tools of similar capabilities.

We note that in order to achieve program speedup, additional program transformations often need to be performed in addition to array data-flow analysis, such as *reduction-loop recognition*, *loop permutation*, *loop fusion*, *advanced induction variable substitution* and so on. Such techniques have been discussed elsewhere and some of them have been implemented in both Polaris [16] and more recently in Panorama. The techniques which are already implemented consume quite insignificant portion of the total analysis and transformation time, since array data-flow analysis is the most time-consuming part. Hence we do not discuss their details in this paper.

The rest of the paper is organized as follows. In Section 2, we present background materials for interprocedural array data-flow analysis and its use for array privatization and loop parallelization. We point out the main factors in such analysis which can potentially slow down the compiler drastically. In Section 3, we present a framework for interprocedural array data-flow analysis based on *guarded array regions*. In Section 4, we discuss several implementation issues. We also briefly discuss how array data-flow analysis can be performed on programs with recursive procedures and dynamic arrays. In Section 5, we discuss the effectiveness and the efficiency of our analysis. Experimental results are reported to show the parallelization capabilities of Panorama and its high time efficiency. We compare related work in Section 6 and conclude in Section 7.

2 Background

In this section, we briefly review the idea of array privatization and give reasons why an aggressive interprocedural array data-flow analysis is needed for this important program transformation.

2.1 Array Privatization

If a variable is modified in different iterations of a loop, writing conflicts result when the iterations are executed by multiple processors. 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

```
A(1)   =                    A(1)  =
DO i = 1, N                 DOALL i = 1, N
DO k = 2, N                   private A1(1:N)
A(k)   =                    A1(1) = A(1)
END DO                      DO k = 2, N
DO k = 2, N                 A1(k) =
   =  A(k-1)                END DO
END DO                      DO k = 2, N
END DO                            = A1(k-1)
                            END DO
                            END DOALL
```

Figure 1: A Simple Example of Array Privatization

serve as a temporary working space within an iteration and the array values in different iterations are unrelated. Array privatization is a technique that creates a distinct copy of an array for each processor such that the storage conflict can be eliminated without violating program semantics. Parallelism in the program is increased. Data access time may also be reduced, since privatized variables can be allocated to local memories. Figure 1 shows a simple example, where the DOALL loop after transformation is to be executed in parallel. Note that the value of A(1) is copied from outside of the DOALL loop since A(1) is not written within the DOALL loop. If the values written to A(k) in the original DO loop are *live* at the end of the loop nest, i.e. the values will be used by statements after the loop nest, additional statements must be inserted in the DOALL loop which, in the last loop iteration, will copy the values of A1(k) to A(k). In this example, we assume A(k) are dead after the loop nest, hence the absence of the copy-out statements.

Practical cases of array privatization can be much more complex than the example in Figure 1. The benefit of such transformation, on the other hand, can be significant. Early experiments with manually-performed program transformations showed that, without array privatization, program execution speed on an Alliant FX/80 machine with 8 vector processors would be slowed down by a factor of 5 for programs MDG, OCEAN, TRACK and TRFD in the well-known Perfect benchmark suite [15]. Recent experiments with automatically transformed codes running on an SGI Challenge multiprocessor show even more striking effects of array privatization on a number of Perfect benchmark programs [16].

2.2 Data Dependence Analysis vs. Array Data-flow Analysis

Conventional data dependence analysis is the predecessor of all current work on array data-flow analysis. In his pioneering work, D.J. Kuck defines flow dependence, anti- dependence and output dependence [26]. While the latter two are due to multi-assignments to the same variable in imperative languages, the flow dependence is defined between two statements, one of which reads the value written by the other. Thus, the original definition of flow dependence 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 dependence is 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 location. Thus, the analysis of all three kinds of data dependences reduces to the problem of memory disambiguation, which is insufficient for array privatization.

Array data-flow analysis refers to computing the flow of values for array elements. For the purpose of array privatization and loop parallelization, the parallelizing compiler needs to establish the fact that, as in the case in Figure 1, no array values are written in one iteration but used in another.

2.3 Interprocedural Analysis

In order to increase the granularity of parallel tasks and hence the benefit of parallel execution, it is important to parallelize loops at outer levels. Unfortunately, such outer-level loops often contain procedure calls. A traditional method to deal with such loops is *in-lining*, which substitutes procedure calls with the bodies of the called procedures. Illinois' Polaris [4], for example, uses this method. Unfortunately, many important compiler transformations increase their consumed time and storage quadratically, or at even higher rates, with the number of operations within individual procedures. Hence, there is a severe limit on the feasible scope of in-lining. It is widely recognized that, for large-scale applications, often a better alternative is to perform interprocedural *summary analysis* instead of in-lining. Interprocedural data dependence analysis has been discussed extensively [21, 24, 31, 40]. In recent years we have seen increased efforts on array data-flow analysis [10, 17, 20, 32, 33, 37, 38, 42]. However, few tools are capable of interprocedural array data-flow analysis without in-lining [10, 20, 23].

2.4 Complications of Array Data-flow Analysis

In reality, a parallelizing compiler not only needs to analyze the effects of procedure calls, but it may also need to analyze relations among symbolic expressions and among branching conditions.

The examples in Figure 2 illustrate such cases. In these three examples, privatizing the array A will make it possible to parallelize the I loops. Figure 2(a) shows a simplified loop from the MDG program (routine interf) [3]. It is a difficult example which requires certain kind of inference between IF conditions. Although both A and B are privatizable, we will discuss A only, as B is a simple case. Suppose that the condition $kc.NE.0$ is false and, as the result, the last loop 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. Condition $kc.NE.0$ being false implies that, within the same iteration of I, the statement $kc = kc + 1$ is not executed. Thus, $B(K).GT.cut2$ is false for all $K = 1, \cdots, 9$ of the first DO loop K. This fact further implies that $B(K+4).GT.cut2$ is false for $K = 2, \cdots, 5$ of the second DO loop 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 2(b) illustrates a simplified version of a segment of the ARC2D program(routine filerx)[3]. The

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

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

(c)

Figure 2: More Complex Examples of Privatizable Arrays.

condition $.NOT.p$ is invariant for DO loop I. As the result, if $A(jmax)$ is not modified in one iteration, thus exposing its use, then $A(jmax)$ 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 upward exposed. Hence, A is privatizable and loop I is a parallel loop. In this example, the IF condition being loop invariant makes sure that there is no loop-carried flow dependence. Otherwise, whether a loop-carried flow dependence exists in Figure 2(b) depends upon the IF condition.

Figure 2(c) shows a simplified version of a segment of the OCEAN program(routine ocean)[3]. Interprocedural analysis is needed for this case. In order to privatize A in the I loop, the compiler must recognize the fact that 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 $A(1 : m)$, so that the use of $A(j)$ must take the value defined in the same iteration of I. This requires to check whether the condition $x > SIZE$ in subroutine *out* can infer the condition $x > SIZE$ in subroutine *in*. For all three examples above, it is necessary to manipulate symbolic operations. Previous and current work suggests that the handling of conditionals, symbolic analysis and interprocedure analysis should be provided in a powerful compiler.

Because array data-flow analysis must be performed over a large scope to deal with the whole set of the subroutines in a program, algorithms for information propagation and for symbolic manipulation must be carefully designed. Otherwise, this analysis will simply be too time-consuming for practical compilers. To handle these issues simultaneously, we have designed a framework which is described next.

3 Array Data-Flow Analysis Based on Guarded Array Regions

In traditional frameworks for data-flow analysis, at each *meet* point of a control flow graph, data-flow information from different control branches is merged under a *meet* operator. Such merged information typically does not distinguish information from different branches. The meet operator can be therefore said to be path-insensitive. As illustrated in the last section, path-sensitive array data-flow information can be critical to the success of array privatization and hence loop parallelization. In this section, we present our path-sensitive analysis that uses conditional summary sets to capture the effect of IF conditions on array accesses. We call the conditional summary sets *guarded array regions* (GAR's).

3.1 Guarded Array Regions

Our basic unit of array reference representation is a *regular array region.*

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 and r_i, $i = 1, \cdots, m$, is a range in the form of $(l : u : s)$, with l, u, s being symbolic expressions. The triple $(l : u : s)$ represents all values from l to u with step s, which 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. □

The regular array region defined above is more restrictive than the original *regular section* proposed by Callahan and Kennedy [6]. The regular array region does not contain any inter-dimensional relationship. This makes set operations simpler. However, a diagonal and a triangular shape of an array cannot be represented exactly. For instance, for an array $A(1 : n, 1 : n)$, a diagonal $A(i, i)$, $i = 1, \ldots, n$ and a triangular $A(i, j)$, $i = 1, \ldots, n, i \leq j$, are approximated by the same regular array region: $A(1 : n, 1 : n)$.

Regular array regions can cover the most frequent cases in real programs and they seem to have an advantage in efficiency when dealing with the common cases. The guards in GAR's (defined below) can be used to describe the more complex array sections, although their primary use is to describe control 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 that the GAR $[P, R] = \Omega$ is *unknown.* Similarly, if either P is $False$ or R is $\emptyset$, we say that $[P, R]$ is $\emptyset$. □

In order to preserve as much precision as possible, we try to avoid marking a whole array region as unknown. If a multi-dimensional array region has only one dimension that is truly unknown, then only that dimension is marked as unknown. Also, if only one item in a range tuple $(l : u : s)$, say u, is unknown, then we write the tuple as $(l : unknown : s)$.

Let a program segment, n, be a piece of code with a unique entry point and a unique exit point. We use results of set operations on GAR's to summarize two essential pieces of array reference information for n which are listed below.

- $UE(n)$: the set of array elements which are upwardly exposed in n if these elements are used in n and they take the values defined outside n.

- $MOD(n)$: the set of array elements written within n.

In addition, the following sets, which are also represented by GAR's, are used to describe array references in a DO loop l with its body denoted by b:

- $UE_i(b)$: the set of the array elements used in an arbitrary iteration i of DO loop l that are upwardly exposed to the entry of the loop body b.

- $UE_i(l)$: the subset of array elements in UE_i(b) which are further upwardly exposed to the entry of the DO loop l.

- $MOD_i(b)$: the set of the array elements written in loop body b for an arbitrary iteration i of DO loop l. Where no confusion results, this may simply be denoted as MOD_i.

- $MOD_i(l)$: the same as $MOD_i(b)$.

- $MOD_{<i}(b)$: the set of the array elements written in all of the iterations prior to an arbitrary iteration i of DO loop l. Where no confusion results, this may simply be denoted as $MOD_{<i}$.

- $MOD_{<i}(l)$: the same as $MOD_{<i}(b)$.

- $MOD_{>i}(b)$: the set of the array elements written in all of the iterations following an arbitrary iteration i of DO loop l. Where no confusion results, this may simply be denoted as $MOD_{>i}$.

- $MOD_{>i}(l)$: the same as $MOD_{>i}(b)$.

Take Figure 2(c) for example. For loop J of subroutine *in*, UE_j is empty and MOD_j equals $[True, B(j)]$. Therefore $MOD_{<j}$ is $[1 < j, B(1 : j-1 : 1)]$ and $MOD_{>j}$ is $[j < mm, B(j+1 : mm : 1)]$. The MOD for the loop J is $[1 \leq mm, A(1 : mm : 1)]$, and hence the MOD of subroutine *in* is $[x \leq SIZE \cap 1 \leq mm, A(1 : mm : 1)]$. Similarly, UE_j for loop J of subroutine *out* is $[True, B(j)]$, and UE for the same loop is $[1 \leq mm, A(1 : mm : 1)]$. Lastly, UE of the subroutine *out* is $[x \leq SIZE \cap 1 \leq mm, A(1 : mm : 1)]$.

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. Next, we will first discuss the operations on array regions, then on GAR's.

3.2 Operations on Regular Array Regions

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 and leave Section 4 for discussion of 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 $D_i = (max(l_i^1, l_i^2) : min(u_i^1, u_i^2) : 1)$. We then have $R_1 \cap R_2$ that equals

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

  Note that we do not keep max and min operators in a regular array region. Therefore, when the relationship of symbolic expressions can not be determined even after a demand-driven symbolic analysis is conducted, we will mark the intersection as unknown.

- $R_1 \cup R_2$:
  Since these regions are symbolic ones, care must be taken to prevent false regions created by union operations. For example, knowing $R_1 = (m : p : 1), R_2 = (p{+}1 : n : 1)$, we have $R_1 \cup R_2 = (m : n : 1)$ if and only if both R_1 and R_2 are valid. This can be guaranteed nicely by imposing validity predicates into guards as we did in [20]. In doing so, the union of two regular regions should be computed without concern for validity of these two regions. Since this introduces additional predicate operations that we try to avoid, we will usually keep the union of two regions without merging them until they, like constant regions, are known to be valid.

- $R_1 - R_2$:
  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. This 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. 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:

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

The following are some examples of difference operations,

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

In order to avoid splitting regions due to difference operations, we routinely defer solving difference operations, using a new data structure called *GARWD* to temporarily represent the difference results. As we shall show later, using GARWD's keeps the summary computation both efficient and exact. GARWD's are defined in the following subsection.

3.3 Operations on GAR's and GARWD's

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 most frequent cases in union operations are of two kinds:

  - 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 can not be safely combined due to the unknown symbolic terms, we 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 discussed previously, $R_1 - R_2$ may be multiple array regions, making the actual result of $T_1 - T_2$ potentially complex. However, as we shall explain via an example, 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 data dependence tests or array privatizability tests. When the difference is not yet solved by the above formula, it is represented by a GARWD.

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 as defined above, while the difference list is a list of GAR's. The *GARWD* set denotes all the members of the source GAR which are not in any GAR on the difference list. It is written as { *source GAR*, *<difference list>* }. □

The following examples show how to use the above formulas:

- $[T, (1:100)] \cap [p, (2:101)] = [p, (2:100)]$
- $[T, (1:50)] \cup [T, (51:100)] = [T, (1:100)]$
  $[T, (1:100)] \cup [p, (1:100)] = [T, (1:100)]$
- $[p, (2:99)] - [T, (1:100)] = [p, ((2:99) - (1:100))] = [p, \emptyset] = \emptyset$
- $[T, (1:100)] - [T, (2:99)]$
  $= \{[T, (1:100)], < [T, (2:99)] >\}$, which is a GARWD. Note that, if we cannot further postpone

```
DO I = 1, M                                   DO I = 1, M
   A(1:N:1)=.      denoted by (3)                    MOD_i: A(1:N:1), A(N2:N2:1)   (3, 2)
   A(N2) = .       denoted by (2)
   ......                                            UE_i: {A(2:N1:1), <(A(1:N:1), A(N2:N2:1)>}   ({1,< 3, 2 >})
     . = A(2:N1:1)    denoted by (1)
ENDDO                                         ENDDO
```

Figure 3: An Example of GARWD's

solving of the above difference, we can solve it to
$[T,((1:100)-(2:99))] = [T,((1) \cup (100))]$
$= [T,(1)] \cup [T,(100)]$

GARWD operations:

Operations between two GARWD's and between a GARWD and a GAR can be easily derived from the above. For example, consider a GARWD gwd=$\{g_1, <g_2>\}$ and a GAR g. The result of subtracting g from gwd is the following:

1. $\{g_3, <g_2>\}$, if $(g_1 - g) = g_3$, or
2. $\{g_1, <g_2>\}$, if $(g - g_2) = \emptyset$, or
3. $\{g_1, <g_2, g>\}$ otherwise.

where g_3 is a single GAR. The first formula is applied if the result of $(g_1 - g)$ is exactly a single GAR g_3. Because g_1 and g may be symbolic, the difference result may not be a single GAR. Hence, we have the third formula. Similarly, the intersection of gwd and g is:

1. $\{g_4, <g_2>\}$, if $(g_1 \cap g) = g_4$, or
2. $\emptyset$, if $(g - g_2) = \emptyset$, or
3. $unknown$ otherwise.

where g_4 is also a single GAR.

The Union of two GARWD's is usually kept in the list, but it can be merged in some cases. Some concrete examples are given below to illustrate the operations on GARWD's:

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

```
... ....
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
```

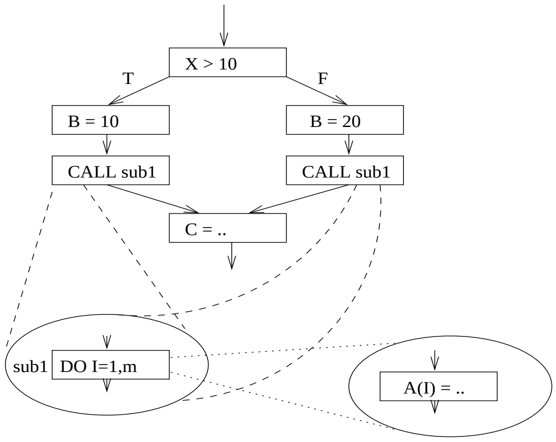


Figure 4: Example of the HSG

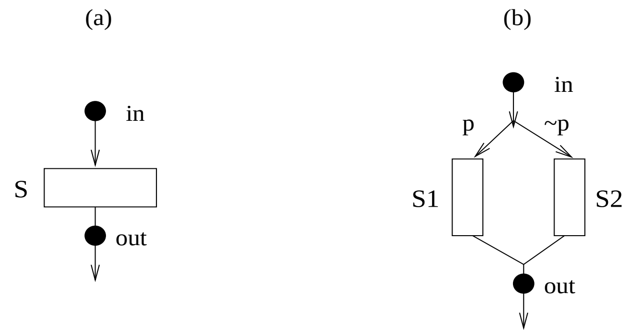
Figure 3 is an example showing the advantage of using GARWD's. The right-hand side is the summary result for the body of the outer loop, where the subscript i in UE_i and in MOD_i indicates that these two sets belong to an arbitrary iteration i. UE_i is represented by a GARWD. For simplicity, we omit the guards whose values are true in the example. To recognize array A as privatizable, we need to prove that no loop-carried data flow exists. The set of all mods within those iterations prior to iteration i, denoted by $MOD_{<i}$, is equal to MOD_i. (In theory, $MOD_{<i} = \phi$ if $i = 1$, which nonetheless does not invalidate the analysis.) Since both GAR's in the $MOD_{<i}$ list are in the difference list of the GARWD for UE_i, it is obvious that the intersection of $MOD_{<i}$ and UE_i is empty, and that therefore array A is privatizable. We implement this by assigning each GAR a unique *region number*, shown in parentheses in Figure 3, which makes intersection a simple integer operation.

As shown above, our difference operations, which are used during the calculation of UE sets, do not result in the loss of information. This helps to improve the effectiveness of our analysis. On the other hand, intersection operations may result in unknown values, due to the intersections of the sets containing unknown symbolic terms. A demand-driven symbolic evaluator is invoked to determine the symbolic values or the relationship between symbolic terms. If the intersection result cannot be determined by the symbolic evaluator, it is marked as unknown.

In our array data-flow framework based on GAR's, intersection operations are performed only at the last step when our analyzer tries to conduct dependence tests and array privatization tests, at the point where a conservative assumption must be made if an intersection result is marked as unknown. The intersection operations, however, are not involved in the propagation of the MOD and UE sets, and therefore they do not affect the accuracy of those sets.

3.4 Computing UE and MOD Sets

The UE and MOD information is propagated backward from the end to the beginning of a routine or a program segment. Through each routine, these two sets are summarized in one pass and the results are



MOD_IN(in) = MOD(S) U MOD_IN(out)

UE_IN(in) = UE(S) U
(UE_IN(out) - MOD(S))

MOD_IN(in) =padd((MOD(S1) U MOD_IN(out)),p)
U padd((MOD(S2) U MOD_IN(out)), ~p)

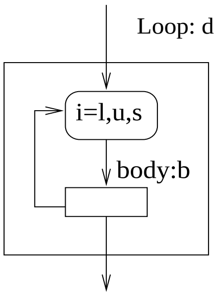
UE_IN(in) =
padd((UE(S1) U (UE_IN(out) - MOD(S1))), p) U
padd((UE(S2) U (UE_IN(out) - MOD(S2))), ~p)

Figure 5: Computing Summary Sets for Basic Control Flow Components

saved. The summary algorithm is invoked on demand for a particular routine, so it will not summarize a routine unless necessary. Parameter mapping and array reshaping are done when the propagation crosses routine boundaries.

To facilitate interprocedural propagation of the summary information, we adopt a *hierarchical supergraph* (HSG) to represent the control flow of the entire program. The HSG augments the *supergraph* proposed by Myers [36] by introducing a hierarchy among nested loops and procedure calls. An HSG contains three kinds of nodes: basic block nodes, loop nodes and call nodes. A DO loop is represented by a loop node which is a compound node whose internal flow subgraph describes the control flow of the loop body. A procedure call site is represented by a call node, which has an outgoing edge pointing to the entry node of the flow subgraph of the called procedure and has an incoming edge from the unique exit node of the called procedure. Due to the nested structures of DO loops and routines, a hierarchy for control flow 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 for parallel task scheduling [25]. Figures 4 shows an example of the HSG. Note that the flow subgraph of a routine is never duplicated for different calls to the same routine, unless multiple versions of the called routine are created by the compiler to enhance its potential parallelism. More details about the HSG and its implementation can be found in reference [20, 18].

During the propagation of the array data-flow information, we use $MOD\_IN(n)$ to represent the array elements that are modified in nodes which are forwardly reachable from n (at the same or lower HSG level as n), and we use $UE\_IN(n)$ to represent the array elements whose values are imported to n and are used in the nodes forwardly reachable from n. Suppose a DO loop l, with its body denoted by b, is represented by a loop node N and the flow subgraph of b has the entry node n. We have $UE_i(b)$ equal to $UE\_IN(n)$ and $UE(N)$ equal to the *expansion* of $UE_i(b)$ (see below). Similarly, we have $MOD_i(b)$



$$MOD(d) = proj(MOD_i(b))$$

$$UE(d) = proj(UE_i(d)), \text{ where } UE_i(d) = UE_i(b) - MOD_{<i}(b)$$

Figure 6: Expansion of Loop Summaries

equal to $MOD\_IN(n)$ and $MOD(N)$ equal to the *expansion* of $MOD_i(b)$. The MOD and MOD_IN sets are represented by a list of GAR's, while the UE and UE_IN sets by a list of GARWD's.

Figure 5 (a) and (b) show how the MOD_IN and UE_IN sets are propagated, in the direction opposite to the control flow, through a basic block S and a flow subgraph for an IF statement (with the then-branch $S1$ and the else-branch $S2$), respectively. During the propagation, variables appearing in certain summary sets may be modified by assignment statements, and therefore their right-hand side expressions substitute for the variables. For simplicity, such variable substitutions are not shown in Figure 5. Figure 5 (b) shows that, when summary sets are propagated to IF branches, IF conditions are put into the guards on each branch, and this is indicated by function *padd*() in the figure.

The whole summary process is quite straightforward, except that the computation of UE sets for loops needs further analysis to support *summary expansion*, as illustrated by Figure 6.

Given a DO loop with index I, $I \in (l, u, s)$, suppose UE_i and MOD_i are already computed for an arbitrary iteration i. We want to calculate UE and MOD sets for the entire I loop, following the formula below:

$$\begin{aligned} MOD &= \Sigma_{i \in (l:u:s)} MOD_i \\ UE &= \Sigma_{i \in (l:u:s)} (UE_i - MOD_{<i}), \\ & \quad MOD_{<i} = \Sigma_{j \in (l:u:s) \wedge (j<i)} MOD_j, \ MOD_{<1} = \phi \end{aligned}$$

The Σ summation above is also called an *expansion* or projection, denoted by $proj()$ in Figure 6, which is used to eliminate i from the summary sets. The UE calculation given above takes two steps. The first step computes $(UE_i - MOD_{<i})$, which represents the set of array elements which are used in iteration i and have been exposed to the outside of the whole I loop. The second step projects the result of Step 1 against the domain of i, i.e. the range $(l : u : s)$, to remove i. The expansion for a list of GAR's and a list of GARWD's consists of the expansion of each GAR and each GARWD in the lists. Since a detailed discussion on expansion would be tedious, we will provide a guideline only in this paper (see Appendix).

```
SUBROUTINE SUB1(A, N2, N3, N4)
   N1 = N2 * N3 + N4
   DO I1=1,100
     DO I2=1,100
       A(N1+I1+I2) = ...                    (S1)
     ENDDO
   ENDDO
END
```

(a)

```
SUBROUTINE SUB2(A, N2, N3, N4, N5, N6)
   N1 = N2 * N3 + N4 * N5
   IF (N1 .GT. N6) THEN
     DO I1=1,N1-1
       A(I1) = ...                          (S2)
     ENDDO
   ENDIF
END
```

(b)

Figure 7: Examples of Symbolic Expressions in Guarded Array Regions

4 Implementation Considerations and Extensions

4.1 Symbolic Analysis

Symbolic analysis handles expressions which involve unknown symbolic terms. It is widely used in symbolic evaluation or abstract interpretation to discover program properties such as values of expressions, relationships between symbolic expressions, etc.. Symbolic analysis requires the ability to represent and manipulate unknown symbolic terms. Among several expression representations, a normal form is often used [7, 9, 22]. The advantage of a normal form is that it gives the same representation for congruent expressions. In addition, symbolic expressions encountered in array data-flow analysis and dependence analysis are mostly integer polynomials. Operations on integer polynomials, such as the comparison of two polynomials, are straightforward. Therefore, we adopt integer polynomials as our representation for symbolic expressions. Our normal form, which is essentially a sum of products, is given below:

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

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

$$t_i = \Sigma_{j=1}^{M_i} p_j, \quad i = 1, \cdots, N \tag{2}$$

$$p_j = c^j \cdot \Pi_{k=1}^{L_j} x_k^j, \quad j = 1, \cdots, M_j \tag{3}$$

where p_j is a product, c^j is an integer constant (possible integer fraction), x_k^j is an integer variable but not an index variable, N is the nesting number 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.

Take the program segments in Figure 7 as examples. For subroutine SUB1, the *MOD* set of statement *S1* contains a single GAR: [True, $A(N1 + I1 + I2)$]. The *MOD* set of DO loop $I2$ contains [True, $A(N1+I1+1 : N1+I1+100)$]. The *MOD* set of DO loop $I1$ contains [True, $A(N1+2 : N1+200)$]. Lastly, the *MOD* set of the whole subroutine contains [True, $A(N2 \cdot N3 + N4 + 2 : N2 \cdot N3 + N4 + 200)$]. For subroutine SUB2, the *MOD* set of statement *S2* contains a single GAR: [True, $A(I1)$]. The *MOD* set of DO loop $I1$ contains [$N1 > 1$, $A(1 : N1 - 1)$]. The *MOD* set of the IF statement contains [$N1 > N6 \wedge N1 > 1$, $A(1 : N1 - 1)$]. Lastly, the *MOD* set of the whole subroutine contains [$N2 \cdot N3 + N4 \cdot N5 > N6 \wedge N2 \cdot N3 + N4 \cdot N5 > 1$, $A(1 : N2 \cdot N3 + N4 \cdot N5 - 1)$].

All expressions e, t_i, and p_j in the above are sorted according to a unique integer key assigned to each variable. Since both M_i and L_j control the complexity of a polynomial, they are chosen as our design parameters. As an example of using M_i and L_j to control the complexity of expressions, e will be a linear expression (affine) if M_i is limited to be 1 and L_j to be zero. By controlling the complexity of expression representations, we can properly control the time complexity of manipulating symbolic expressions.

Symbolic operations such as additions, subtractions, multiplications, and divisions by an integer constant are provided as library functions. In addition, a simple demand-driven symbolic evaluation scheme is implemented. It propagates an expression upwards along a control flow graph until the value of expression is known or the predefined propagation limit is reached.

4.2 Range Operations

In this subsection, we give a detailed discussion of range operations for step values other than 1. 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 should be solved, otherwise the unknown is usually returned as the result. 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
     $$\begin{aligned} r_1 - r_2 \quad = \quad & [l_1 \leq max(l_1, l_2) - s_1, (l_1 : max(l_1, l_2) - s_1 : s_1)] \cup \\ & [min(u_1, u_2) + s_1 \leq u_1, (min(u_1, u_2) + s_1 : u_1 : s_1)] \end{aligned}$$
     where $max(l_1, l_2) = l_2, min(u_1, u_2) = u_2$ because $r_2 \subseteq r_1$.
   - 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, $r_1 \cup r_2 = [True, (min(l_1, l_2) : max(u_1, u_2) : s_1)]$, assuming that r_1 and r_2 are both valid. If it is unknown at this moment whether both are valid, we do not combine them.
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 cannot be combined into one range and must be maintained as a list of ranges. For the special case that $|l_1 - l_2| = |u_1 - u_2| = 1$ and $s_1 = s_2 = 2$, we have
   $$r_1 \cup r_2 = (min(l_1, l_2) : max(u_1, u_2) : 1).$$
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 to see if r_2 covers r_1. If so, we have $r_1 \cap r_2 = r_1$, $r_1 - r_2 = \emptyset$, and $r_1 \cup r_2 = r_2$.

5. In all other cases, the result of the intersection is marked as unknown. The difference is kept in a difference list at the level of the GARWD's, and the union remains a list of the two ranges.

4.3 Extensions to Recursive Calls and Dynamic Arrays

Programming languages such as Fortran 90 and C permit recursive procedure calls and dynamically allocated data structures. In this subsection, we briefly discuss how array data-flow analysis can be performed in the presence of recursive calls and dynamic arrays.

Recursive calls can be treated in array data-flow analysis essentially the same way as in array data dependence analysis [30]. A recursive procedure calls itself either directly or indirectly, which forms cycles in the *call graph* of the whole program. A proper order must be established for the traversal of the call graph. First, all *maximum strongly-connected components* (MSC's) must be identified in the call graph. Each MSC is then reduced to a single *condensed node* and the call graph is reduced to an acyclic graph. Array data flow is then analyzed by traversing the reduced graph in a reversed *topological order*. When a condensed node (i.e. an MSC) is visited, a proper order is established among all members in the MSC for an iterative traversal. For each member procedure, the sets of modified and used array regions (with guards) that are visible to its callers must be summarized respectively, by iterating over calling cycles. If the MSC is a simple cycle, which is a common case in practical programs, the compiler can determine whether the visible array regions of each member procedure grow through recursion or not, after analyzing that procedure twice. If a region grows in a certain array dimension during recursive calls, then a conservative estimate should be made for that dimension. In the worst case, for example, the range of modification or use in that array dimension can be marked as *unknown*. A more complex MSC requires a more complex traversal order [30].

Dynamically allocated arrays can be summarized essentially the same way as static arrays. The main difference is that, during the backward propagation of array regions (with guards) through the control flow graph, i.e. the HSG in this paper, if the current node contains a statement that allocates a dynamic array, then all *UE* sets and *MOD* sets for that array are killed beyond this node.

The discussion above is based on the assumption that no *true aliasing* exists in each procedure, i.e., references to different variable names must access different memory locations if either reference is a write. This assumption is true for Fortran 90 and Fortran 77 programs, but may be false for C programs. Before performing array data-flow analysis on C programs, alias analysis must first be performed. Alias analysis has been studied extensively in recent literature [8, 11, 14, 27, 28, 39, 44, 45].

5 Effectiveness and Efficiency

In this section, we first discuss how GAR's are used for array privatization and loop parallelization. We then present experimental results to show the effectiveness and efficiency of array data-flow analysis.

5.1 Array Privatization and Loop Parallelization

An array A is a privatization candidate in a loop L if its elements are overwritten in different iterations of L (see [29]). 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(c) again. $UE_i = \emptyset$ and $mod_{<i} = [x \le SIZE \wedge 1 < m \wedge 1 < i, A(1 : m : 1)]$. Hence, $MOD_{<i} \cap UE_i = MOD_{<i} \cap \emptyset = \emptyset$, so A is privatizable within loop I. As another example, let us look at Figure 2(b). Since MOD_i is not loop-variant, we have $MOD_{<i} = MOD_i$. Hence, $MOD_i \cap MOD_{<i}$ is not empty and array A is a privatization candidate. Furthermore,

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

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

As we explained in Section 2.1, copy-in and copy-out statements sometimes need to be inserted in order to preserve program correctness. The general rules are (1) upwardly exposed array elements must be copied in; and (2) *live* array elements must be copied-out. We have already discussed the determination of upwardly exposed array elements. We currently perform a conservative liveness analysis proposed in [29].

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:

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

- **loop-carried anti- dependences:** Suppose we have already determined that there exist no loop-carried output dependences, then loop-carried anti-dependences exist if and only if $UE_i \cap MOD_{>i} \neq \emptyset$. (If loop-carried anti-dependences were to be considered separately, then UE_i in the above formula should be replaced by DE_i, where DE_i stands for the *downwardly exposed* use set of iteration i.)

Take output dependences for example. In Figure 7(a), MOD_i of DO loop $I2$ contains a single GAR: [True, $A(N1+I1+i)$]. $MOD_{<i}$ contains [$i>1$, $A(N1+I1+1:N1+I1+i-1)$] and $MOD_{>i}$ contains [$i<100$, $A(N1+I1+i+1:N1+I1+100)$]. Loop-carried output dependences do not exist for DO loop I2 because $MOD_i \cap (MOD_{<i} \cup MOD_{>i}) = \emptyset$. In contrast, for DO loop $I1$, MOD_i contains [True, $A(N1+i+1:N1+i+100)$]. $MOD_{<i}$ contains [$i>1$, $A(N1+2:N1+i+99)$]. Loop-carried output dependences exist for DO loop $I1$ because $MOD_i \cap MOD_{<i} \neq \emptyset$. Note that if an array is privatized, then no loop-carried output dependences exist between the write references to private copies of the same array.

5.2 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 analyzer [35]. It roughly consists of the phases of parsing, building a hierarchical supergraph (HSG) and the interprocedural scalar UD/DU chains [1], performing conventional data dependence tests, array data-flow analysis and other advanced analyses, and parallel code generation.

Table 1 shows the Fortran loops in the Perfect benchmark suite which should be parallelizable after array privatization and after necessary transformations such as induction variable substitution, parallel reduction, and event synchronization placement. This table also marks which loops require symbolic analysis, predicate analysis and interprocedural analysis, respectively. (The details of privatizable arrays in these loops can be found in [18].)

Columns 4 and 5 mark those loops that can be parallelized by Polaris (Version 1.5) and by Panorama, respectively. Only one loop (interf/1000) is parallelized by Polaris but not by Panorama, because one of the privatizable arrays is not recognized as such. To privatize this array requires implementation of a special pattern matching which is not done in Panorama. On the other hand, Panorama parallelizes several loops that cannot be parallelized by Polaris.

Table 2 compares the speedup of the programs selected from Table 1, parallelized by Polaris and by Panorama, respectively. Only those programs parallelizable by either or both of the tools are selected. The speedup numbers are computed by dividing the real execution time of the sequential codes divided by the real execution time of the parallelized codes, running on an SGI Challenge multiprocessor with four 196MHZ R10000 CPU's and 1024 MB memory. On average, the speedups are comparable between Polaris-parallelized codes and Panorama-parallelized codes. Note that the speedup numbers may be

Table 1: Parallelizable Loops in the Perfect benchmark suite and the Required Privatization Techniques

Program	Routine /Loop	SA	PA	IA	Parallel	
					Polaris	Panorama
ADM	dcdtz/40	Yes	Yes	Yes		√
	dtdtz/40	Yes	Yes	Yes		√
	dudtz/40	Yes	Yes	Yes		√
	dvdtz/40	Yes	Yes	Yes		√
	dkzmh/30	Yes	No	Yes		√
	dkzmh/60	Yes	No	Yes		√
	wcont/40	Yes	No	Yes		√
TRACK	nlfilt/300	No	No	Yes		
MDG	interf/1000	Yes	Yes	Yes	√	
	poteng/2000	No	No	Yes	√	√
TRFD	olda/100	Yes	No	No	√	√
	olda/300	Yes	No	No	√	√
OCEAN	ocean/270	Yes	No	Yes		√
	ocean/480	Yes	No	Yes		
	ocean/500	Yes	No	Yes		√
ARC2D	filerx/15	Yes	Yes	Yes		√
	filery/39	Yes	No	No	√	√
	stepfx/300	Yes	No	Yes		
	stepfy/420	Yes	No	Yes		
QCD	measur/3	Yes	Yes	Yes		
BDNA	actfor/240	No	Yes	No	√	√
FLO52	step/20	Yes	No	No	√	√
MG3D	migrat/200	Yes	No	Yes		
SPEC77	gwater/1000	No	No	Yes		
	gloop/1000	No	No	Yes		
Total		80%	32%	80%	7	16

SA: Symbolic Analysis. PA: Predicate Analysis. IA: Interprocedural Analysis.

further improved by a number of recently discovered memory-efficiency enhancement techniques. These techniques are not implemented in the versions of Polaris and Panorama used for this experiment.

Table 3 shows wall-clock time spent on the main parts of Panorama. In Table 3, "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.)

The column "HSG & DOALL Checking" is the time taken to build the HSG, UD/DU chains, and conventional DOALL checking. The column "Array Summary" refers to our array data-flow analysis which is applied only to loops whose parallelizability cannot be determined by the conventional DOALL tests. Figure 8 shows the percentage of time spent by the array data-flow analysis and the rest of Panorama. Even though the time percentage of array data-flow analysis is high (about 38% on average), the total execution time is small (31 seconds maximum). To get a perspective of the overhead of our

Table 2: Speedup Comparison between Polaris and Panorama (with 4 R10000 CPU's).

Program	Speedup by Polaris	Speedup by Panorama
ADM	1.1	1.5
MDG	2.0	1.5
BDNA	1.2	1.2
OCEAN	1.2	1.7
ARC2D	2.1	2.2
FLO52	1.6	2.1
TRFD	2.2	2.1

interprocedural analysis, the last column, 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.

Table 4 lists the analysis time of Polaris alongside of that of Panorama (which includes all three times of parsing instead of just one as in Table 3). It is difficult to provide an absolutely fair comparison. So, these two sets of numbers are listed together to provide a perspective. The timing of Polaris (Version 1.5) is measured without the passes after array privatization and dependence tests. (We did not list the timing results of SUIF, because SUIF's current public version does not perform array data-flow analysis and no such timing results are publically available.) Both Panorama and Polaris are compiled by the GNU gcc/g++ compiler with the -O optimization level. The time was measured by `gettimeofday()` and is elapsed wall-clock time. When using a SGI Challenge machine, which has a large memory, the time gap between Polaris and Panorama is reduced. This is probably because Polaris is written in C++ with a huge executable image. The size of its executable image is about 14MB, while Panorama, written in C, has an executable image of 1.1MB. Even with a memory size as large as 1GB, Panorama is still faster than Polaris by one or two orders of magnitude.

5.3 Summary vs. In-lining

We believe that several design choices contribute to the efficiency of Panorama. In the next subsections, we present some of these choices made in Panorama.

The foremost reason seems to be that Panorama computes interprocedural summary without in-lining the routine bodies as Polaris does. If a subroutine is called in several places in the program, in-lining causes the subroutine body to be analyzed several times, while Panorama only needs to summarize each subroutine once. The summary result is later mapped to different call sites. Moreover, for data dependence tests involving call statements, Panorama uses the summarized array region information, while Polaris performs data dependences between every pair of array references in the loop body after in-lining. Since the time complexity of data dependence tests is $O(n^2)$, where n is the number of individual references being tested, in-lining can significantly increase the time for dependence testing. In our experiments with Polaris, we limit the number of in-lined executable statements to 50, a default

Table 3: Analysis Time (in seconds) Distribution[1]

Program	Parsing	HSG & Tradi. Analysis	Array Summary	Code Generation	Total	F77 -O
ADM	3.63	12.68	11.76	3.53	31.60	54.1
QCD	1.04	3.71	3.04	1.22	9.01	20.3
MDG	0.82	2.58	2.11	0.77	6.28	12.3
TRACK	1.58	4.07	10.81	1.29	17.76	17.5
BDNA	2.41	7.41	3.80	2.45	16.06	45.2
OCEAN	1.37	8.49	3.31	1.35	14.53	39.3
DYFESM	3.77	6.04	2.26	2.48	14.56	20.2
MG3D	1.67	7.46	14.87	1.70	25.71	34.0
ARC2D	2.46	6.24	10.14	1.96	20.81	37.7
FLO52	1.49	4.78	0.50	1.22	7.99	30.3
TRFD	0.54	0.70	0.48	0.18	1.90	7.2
SPEC77	3.77	7.94	7.23	2.22	21.17	31.7
Total	24.56	72.1	70.31	20.37	187.38	349.8

1: Timing is measured on SGI Indy workstations with 134MHz MIPS R4600 CPU and 64 MB memory.

Table 4: Elapsed Analysis Time (in seconds)

Program	#Lines excl. comments	SGI Challenge		SGI Indy[2]	
		Panorama	Polaris	Panorama	Polaris
ADM	4296	17.03	435	38.80	2601
QCD	1710	5.51	311	11.09	1065
MDG	935	3.02	123	7.90	551
TRACK	1806	8.05	81	20.96	335
BDNA	3663	8.32	218	20.87	840
OCEAN	1917	8.70	333	18.2	1801
DYFESM	3386	6.95	121	22	533
MG3D	2505	10.49	3145	29	*
ARC2D	2433	8.91	88	25.81	391
FLO52	1758	3.56	71	10.99	299
TRFD	417	1.05	62	2.98	290
SPEC77	3291	8.94	1074	28.67	5376

[1]SGI Challenge with 1024MB memory and 196MHZ R10000 CPU. [2]SGI Indy with 134MHz MIPS R4600 CPU and 64 MB memory. [3] '*' means Polaris takes longer than four hours.

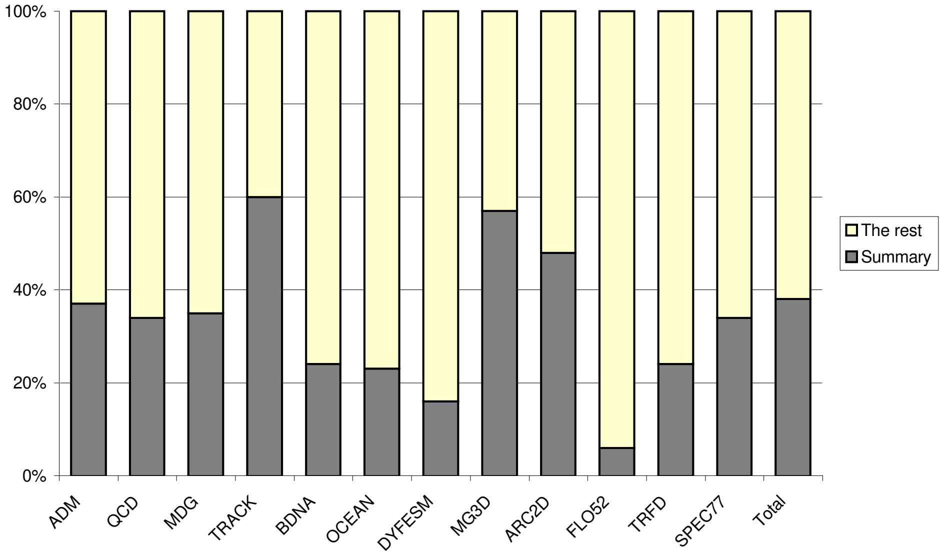


Figure 8: Time percentage of array data-flow summary

value used by Polaris. With this modest number, data dependence tests still account for about 30% of the total time.

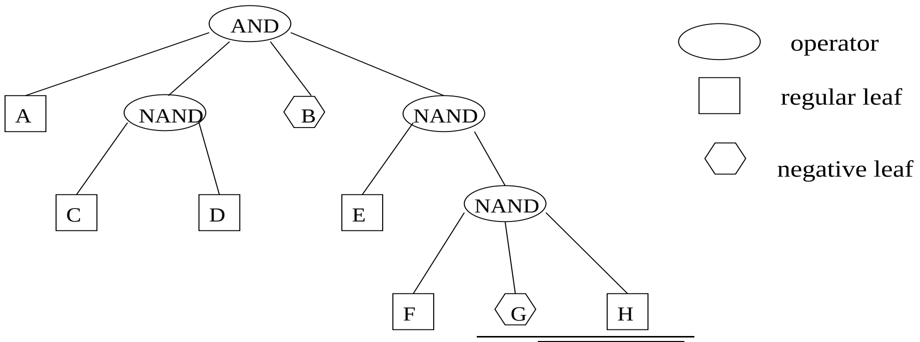
We believe that another important reason for Panorama's efficiency is its efficient computation and propagation of the summary sets. Two design issues are particularly noteworthy, namely, the handling of predicates and the difference set operations. Next, we discuss these issues in more details.

5.4 Efficient Handling of Predicates

General predicate operations are expensive, so compilers often do not perform them. In fact, the majority of predicate-handling required for our array data-flow analysis involves simple operations such as checking to see if two predicates are identical, if they are loop-independent, and if they contain indices and affect shapes or sizes of array regions. These can be implemented rather efficiently.

A canonical normal form is used to represent the predicates. Pattern-matching under a normal form is easier than under arbitrary forms. Both the conjunctive normal form (CNF) and the disjunctive normal form (DNF) have been widely used in program analysis [7, 9]. These cited works show that negation operations are expensive with both CNF and DNF. This fact was also confirmed by our previous experiments using CNF [20]. Negation operations occur not only due to ELSE branches, but also due to GAR and GARWD operations elsewhere. Hence, we design a new normal form such that negation operations can often be avoided.

We use a hierarchical approach to predicate handling. A predicate is represented by a high level 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 nodes of V are NAND operators except for 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



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

Figure 9: High level representation of predicates

a predicate such as an IF condition, while a negative leaf node represents the negation of a predicate. Theoretically, this representation is not a normal form because two identical predicates may have different predicate trees, which may render pattern-matching unsuccessful. We, however, believe that such cases are rare and that they happen only when the program is extremely complicated. Figure 9 shows a PT.

Each leaf (regular or negative) is a token which represents a basic predicate such as an IF condition or a DO condition in the program. At this level, we keep a basic predicate as a unit and do not split it. The predicate operations are based only on these tokens and do not check the details within these basic predicates. Negation of a predicate tree is simple this way. A NAND operation, shown in Figure 10, may either increase or decrease by one level in a predicate tree according to the shape of the predicate tree. 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). AND and OR operations are also easily handled, as shown in Figure 10. We use a unique token for each basic predicate so that simple and common cases can be easily handled without checking the contents of the predicates. The content of each predicate is represented in CNF and is examined when necessary.

Table 5 lists several key parameters, the total number of arrays summarized, the average length of a MOD set (column "Ave # GAR's"), the average length of a UE set (column "Ave # GARWD's"), and some data concerning difference and predicate operations. The total number of arrays summarized given in the table is the sum of the number of arrays summarized in each loop nest, and an array that appears in two disjoint loop nests is counted twice. Since the time for set operations is proportional to the square of the length of MOD and UE lists, it is important that these lists are short. It is encouraging to see that they are indeed short in the benchmark application programs.

Columns 7 and 8 (marked "High" and "Low") in Table 5 show that over 95% of the total predicate operations are the high level ones, where a negation or a binary predicate operation on two basic predicates is counted as one operation. These numbers are dependent on the strategy used to handle the predicates. Currently, we defer the checking of predicate contents until the last step. As a result, only a few low level

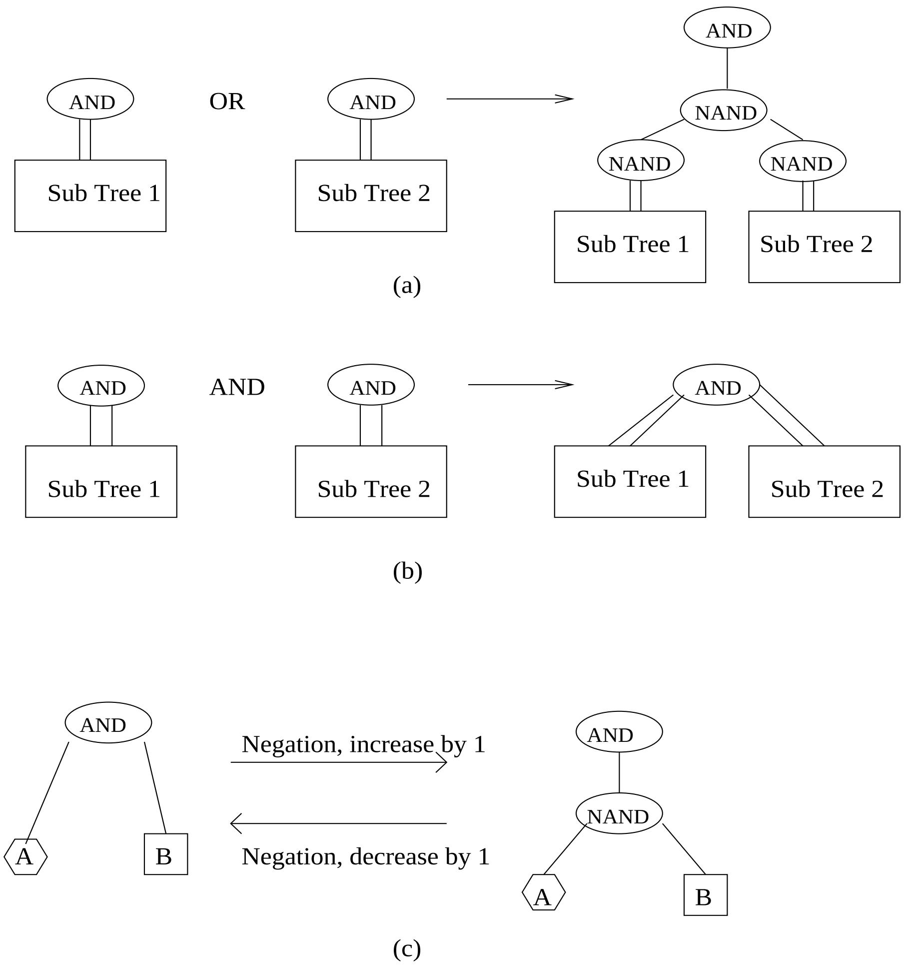


Figure 10: Predicate operations

Table 5: Measurement of Key Parameters

Program	# Array Summarized	Ave # GAR's	Ave # GARWD's	Difference Ops		# Predicate Ops	
				Total	Reduced	High	Low
ADM	731	1.81	1.50	502	80	8100	285
QCD	414	1.41	1.27	512	41	4803	41
MDG	104	1.25	1.56	180	59	1198	18
TRACK	134	1.37	1.60	249	1	9028	11
BDNA	285	1.27	1.43	267	3	3805	4
OCEAN	96	1.72	1.53	246	19	458	36
DYFESM	342	1.27	1.24	128	17	1696	47
MG3D	385	2.79	2.62	135	65	5063	22
ARC2D	633	1.65	1.34	1055	2	1942	60
FLO52	32	1.47	1.20	9	0	310	0
TRFD	81	1.29	1.38	36	2	182	18
SPEC77	774	1.33	1.24	356	25	5806	76
Total	4011	1.55	1.49	3675	314	42391	618

predicate operations are needed. Our results show that this strategy works well for array privatization, since almost all privatizable arrays in our tested programs can be recognized. Some cases, such as those that need to handle guards containing loop indices, do need low level predicate operations. The hierarchical representation scheme serves well.

5.5 Reducing Unnecessary Difference Operations

We do not solve the difference of $T_1 - T_2$ using the general formula presented in Section 2 unless the result is a single GAR. When the difference cannot be simplified to a single GAR, the difference is represented by a GARWD instead of by a union of GAR's, as implied by that formula. This strategy postpones the expensive and complex difference operations until they are absolutely necessary, and it avoids propagating a relatively complex list of GAR's. For example, let a GARWD G_1 be $\{(1:m), <(k:n), (2:n1)>\}$ and G_2 be $(1:m)$. We have $G_1 - G_2 = \phi$, and two difference operations represented in G_1 are reduced (i.e., there is no need to perform them). In Table 5, the total number of difference operations and the total number of reduced difference operations are illustrated in columns 5 and 6, respectively. Although difference operations are reduced by only about 9% on average, the reduction is dramatic for some programs: it is by one third for MDG and by half for MG3D.

Let us use the example in Figure 2(b) to further illustrate the significance of delayed difference operations. A simplified control flow graph of the body of the outer loop is shown in Figure 11. Suppose

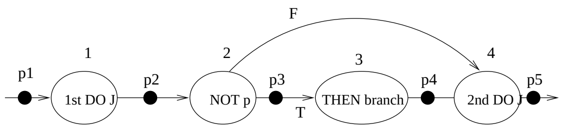


Figure 11: The HSG of the Body of the Outer Loop for Figure

that each node has been summarized and that the summary results are listed below:

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

Following the description given in Section 3.4, we will propagate the summary sets of each node in the following steps to get the summary sets for the body of the outer loop.

1. $MOD\_IN(p4) = MOD(4) = \emptyset$

   $UE\_IN(p4) = UE(4) = [T,(jlow:jup)] \cup [T,(jmax)]$

2. $MOD\_IN(p3) = MOD(3) \cup MOD\_IN(p4)$

   $= [T,(jmax)]$

   $UE\_IN(p3) = UE(3) \cup (UE\_IN(p4) - MOD(3))$

   $= \{[T,(jlow:jup)], < [T,(jmax)] >\}$

   This difference operation is kept in the GARWD and will be reduced at step 4.

3. $MOD\_IN(p2) = [\overline{p},(jmax)]$

   $UE\_IN(p2) = \{[\overline{p},(jlow:jup)], < [\overline{p},(jmax)] >\}$

   $\cup [p,(jlow:jup)] \cup [p,(jmax)]$

   In the above, $\overline{p}$ is inserted into the guards of the GAR's, which are propagated through the TRUE edge, and p is then inserted into the guards propagated through the FALSE edge.

4. $MOD\_IN(p1) = [T,(jlow:jup)] \cup [\overline{p},(jmax)]$

   $UE\_IN(p1) = UE\_IN(p2) - MOD(1)$

   $= \{[p,(jmax)], < [T,(jlow:jup)] >\}$

   At this step, the computation of $UE\_IN(p1)$ removes one difference operation because $(\{[\overline{p},(jlow:jup)], < [\overline{p},(jmax)] >\} - [T,(jlow:jup)])$ is equal to $\emptyset$. In other words, there is no need to perform the difference operation represented by GARWD $\{[\overline{p},(jlow:jup)], < [\overline{p},(jmax)] >\}$. An advantage of the GARWD representation is that a difference can be postponed rather than always performed. Without using a GARWD, the difference operation at step 2 always has to be performed, which should not be necessary and which thus increases execution time.

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

$$\begin{aligned} MOD_i &= MOD\_IN(p1) = [T, (jlow : jup)] \cup [\overline{p}, (jmax)] \\ UE_i &= UE\_IN(p1) = \\ &\quad \{[p, (jmax)], < [T, (jlow : jup)] >\} \end{aligned}$$

To determine if array A is privatizable, we need to prove that there exists no loop-carried flow dependence for A. We first calculate $MOD_{<i}$, the set of array elements written in iterations prior to iteration i, giving us $MOD_{<i} = MOD_i$. The intersection of $MOD_{<i}$ and UE_i is conducted by two intersections, each of which is formed on one mod component from $MOD_{<i}$ and UE_i respectively. The first mod, $[T, (jlow : jup)]$, appears in the difference list of UE_i, and thus the result is obviously empty. Similarly, the intersection of $[p, (jmax)]$ and the second mod, $[\overline{p}, (jmax)]$, is empty because their guards are contradictory. Because the intersection of $MOD_{<i}$ and UE_i is empty, array A is privatizable. In both intersections, we avoid performing the difference operation in UE_i, and therefore improve efficiency.

6 Related Work

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 presented efficiency data. One school of thought attempts to gather flow information for each array element and to acquire an *exact* array data-flow analysis. This is usually done by solving a system of equalities and inequalities. Feautrier [17] calculates the *source function* to indicate detailed flow information. Maydan et al. [33, 34] simplify Feautrier's method by using a Last-Write-Tree(LWT). Duesterwald et al. [12] compute the dependence distance for each reaching definition within a loop. Pugh and Wonnacott [37] use a set of constraints to describe array data-flow problems and solve them basically by the Fourier-Motzkin variable elimination. Maslov [32], as well as Pugh and Wonnacott [37], also extend the previous work in this category by handling certain IF conditions. Generally, these approaches are intraprocedural and do not seem easily extended interprocedurally. The other group analyzes a set of array elements instead of individual array elements. Early work uses *regular sections* [6, 24], convex regions [40, 41], data access descriptors [2], etc. to summarize MOD/USE sets of array accesses. They are not array data-flow analyses. Recently, array data-flow analyses based on these sets were proposed (Gross and Steenkiste [19], Rosene [38], Li [29], Tu and Padua [43], Creusillet and Irigoin [10], and M. Hall et al. [21]). Of these, ours is the only one using conditional regions (GAR's), even though some do handle IF conditions using other approaches. Although the second group does not provide as many details about reaching-definitions as the first group, it handles complex program constructs better and can be easily performed interprocedurally.

Array data-flow summary, as a part of the second group mentioned above, has been a focus in the parallelizing compiler area. The most essential information in array data-flow summary is the upwardly exposed use set. These summary approaches can be compared in two aspects: set representation and path sensitivity. For set representation, convex regions are highest in precision, but they are also expensive because of their complex representation. Bounded regular sections (or regular sections) have the simplest representation, and thus are most inexpensive. Early work tried to use a single regular section or a single convex region to summarize one array. Obviously, a single set can potentially lose information, and it may be ineffective in some cases. Tu and Padua [43], and Creusillet and Irigoin [10] seem to use a single regular section and a single convex region, respectively. M. Hall et al. [21] use a list of convex regions to summarize all the references of an array. It is unclear if this representation is more precise than a list of regular sections, upon which our approach is based.

Regarding path sensitivity, the commonality of these previous methods is that they do not distinguish summary sets of different control flow paths. Therefore, these methods are called path-insensitive, and have been shown to be inadequate in real programs. Our approach, as far as we know, is the only path-sensitive array data-flow summary approach in the parallelizing compiler area. It distinguishes summary information from different paths by putting IF conditions into guards. Some other approaches do handle IF conditions, but not in the context of array data-flow summary.

7 Conclusion

In this paper, we have presented an array data-flow analysis which handles interprocedural, symbolic, and predicate analyses all together. The analysis is shown via experiments to be quite effective for program parallelization. Important design decisions are made such that the analysis can be performed efficiently. Our hierarchical predicate handling scheme turns out to serve very well. Many predicate operations can be performed at high levels, avoiding expensive low-level operations. The new data structure, GARWD (i.e. guarded array regions with a difference list), reduces expensive set-difference operations by up to 50% for a few programs, although the reduction is unimpressive for other programs. Another important finding is that the MOD lists and the UE lists can be kept rather short, thus reducing set operation time.

As far as we know, this is the first time the efficiency issue has been addressed and data presented for such a powerful analysis. We believe it is important to continue exploring the efficiency issue, because unless interprocedural array data-flow analysis can be performed reasonably fast, its adoption in real programming world would be unlikely. With continued advances of parallelizing compiler techniques, we hope that fully or partially automatic parallelization will provide a viable methodology for machine-independent parallel programming.

References

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

[2] 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.

[3] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger and Peng Tu. Parallel Programming with Polaris. *IEEE Computer*, 28(11):78–82, December 1996.

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

[6] 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.

[7] T.E. Cheatham Jr., G.H. Holloway, and J.A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Trans.on Software Engineering*, 5(4):402–417, July 1979.

[8] J.-D. Choi, M. Burke and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *Proc. of the 20th Annual ACM Symposium on Principles of Programming languages*, pp. 232–245, January 1993.

[9] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *The Journal of Systems and Software*, 5(1):15–35, 1985.

[10] Béatrice Creusillet and F. Irigoin. Interprocedural array region analyses. *Int. Journal of Parallel Programming*, 24(6):513–546, December 1996.

[11] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 230–241, June 1994.

[12] 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.

[13] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. Technical Report TR 1392, CSRD, University of Illinois at Urbana-Champaign, November 1994.

[14] M. Emami, R. Ghiya and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 242–256.

[15] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four perfect-benchmark programs. In *Proc. of the 4-th Workshop on Languages and Compilers for Parallel Computing.* also available as CSRD Tech. Rept No. 1114, Univ. of Illinois at Urbana-Champaign, August 1991.

[16] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the Perfect Benchmarks. *IEEE Trans. on Parallel and Distributed Systems*, 9(1):5–23, Jan. 1998.

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

[18] J. Gu *Interprocedural Array Data-Flow Analysis*, Ph.D. thesis, University of Minnesota, December, 1997.

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

[20] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing*, December 1995.

[21] M.W. Hall, B.R. Murphy, S.P. Amarasinghe, S.-W. Liao, and M.S. Lam. Interprocedural analysis for parallelization. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing, No. 1033, In Lecture Notes in Computer Science, Springer-Verlag, Berlin*, pages 61–80, August 1995.

[22] M. R. Haghighat and C. D. Polychronopoulos. Symbolic dependence analysis for parallelizing compilers. Technical Report CSRD Report No. 1355, Center for Supercomputing Research and Development, University of Illinois, 1994.

[23] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF Compiler. *IEEE Computer*, 28(11): 84–89, December 1996.

[24] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[25] F. Irigoin, P. Jouvelot and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. *Proc. of ACM International Conference on Supercomputing*, 1991, pp. 244-251.

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

[27] W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.

[28] W. Landi, B.G. Ryder and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 56–67, June 1993.

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

[30] Z. Li and P.-C. Yew. Interprocedural analysis for parallel computing. *Proc. of the 1988 International Conference on Parallel Processing*, August, 1988, St. Charles, Illinois, pp. 221–228.

[31] Z. Li and P.-C. Yew. Program parallelization with interprocedural analysis. *Journal of Supercomputing*, 2(2):225-244, Oct. 1988.

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

[33] 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.

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

[35] T. Nguyen, J. Gu, and Z. Li. An interprocedural parallelizing compiler and its support for memory hierarchy research. In *Lecture Notes in Computer Science 1033: 8th International Workshop on Languages and Compilers for Parallel Computing*, pages 96–110, Columbus, Ohio, August 1995. Springer-Verlag.

[36] E.W. Myers. A precise interprocedural data-flow algorithm. *Proc. of 8th Annual ACM Symposium on Principles of Programming Languages*, January 1981, pp. 219-230.

[37] 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.

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

[39] E. Ruf. Context-insensitive alias analysis reconsidered. *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13-31, June 1995.

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

[41] 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.

[42] P. Tu and D. Padua. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In *International Conference on Supercomputing*, pages 414–423, July 1995.

[43] 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.

[44] R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12, June 1995.

[45] S. Zhang and B.G. Ryder and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. *Proc. of the 4th Symposium on the Foundations fo Software Engineering*, October, 1996.

Appendix: Expansion of Loop Summaries.

In the following, we present a guideline for computing the expansion of loop summaries introduced in Section 3.4. First, for a GAR Q, $proj(Q)$ is obtained by the following steps:

1. If i appears in the guard of a GAR, we remove the predicate components which involve i from the guard, and we use such components to derive a new domain of i. Suppose that i in the guard can be solved and represented as $i \in (l' : u')$. The new domain of i becomes

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

   which simplifies to $(max(l', l) : min(u', u))$ for $s = 1$.

   For example, given $i \in (2 : 100 : 2)$ and GAR $[5 \leq i, A(i)]$, we remove the relational expression $5 \leq i$ from the guard and form the new domain of i: $(\lceil \frac{max(5,2)-2}{2} \rceil \cdot 2 + 2 : 100 : 2) = (6 : 100 : 2)$. Hence, the projection will be completed by expanding $[T, A(i)]$, $i \in (6 : 100 : 2)$, whose result is $[T, A(6 : 100 : 2)]$.

2. Suppose that i appears in only one dimension of Q. If the result of substituting $l \leq i \leq u$, 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 $(l'' : u'' : s'')$. For example, the range triple $(i : i : 1)$ becomes $(l : u : 1)$.

3. If, in the above, the result of substituting $l \leq i \leq u$ into the old range can no longer be represented by a range, or if i appears in more than one dimension of Q, then these dimensions are marked as unknown. (Tighter approximation is possible for special cases, but we will not discuss it in this paper.)

For the expansion of a GARWD, we have the following:

1. For a GARWD, if its difference list and its source GAR cannot be expanded separately, then we must solve the difference list first, invoking the symbolic evaluator if necessary. If the difference list cannot be solved, the expansion result is marked as unknown.

2. The computation of $(UE_i - MOD_{<i})$ and its expansion can be done without expanding MOD_i to $MOD_{<i}$ first. Instead, $(UE_i - MOD_{<i})$ is evaluated to $UE_{i'}$ with a new index variable i'. Consider a special case in which $UE_i = \{A(I + m), <>\}$ and $MOD_i = A(I + n)$. We can formulate

   $$\begin{array}{l} (UE_i - MOD_{<i}), i \in (l : u) = \\ \quad \left\{ \begin{array}{ll} UE_{i'},\ i' \in (l : l + (m - n) - 1); & (m - n) > 0 \\ UE_{i'},\ i' \in (l : u); & (m - n) \leq 0. \end{array} \right. \end{array}$$

   As a concrete example, suppose we have $i \in (2 : 99)$, $MOD_i = [T, A(i + 1)]$, and $UE_i = [T, A(i)]$, which satisfies $(m - n) > 0$ in the above. The set $(UE_i - MOD_{<i})$, with $i \in (2 : 99)$, should equal set $UE_{i'}$, $i' \in (2 : 2)$.

Suppose, however, that MOD_i is $[T, A(i-1)]$. The case of $(m-n) \leq 0$ applies instead. The set $(UE_i - MOD_{<i})$, with $i \in (2:99)$, equals $UE_{i'}$, with $i' \in (2:99)$.

In this paper, we leave out the general discussion on the short-cut computation illustrated above.