Experience with Efficient Array Data Flow Analysis for Array Privatization[1]

Junjie Gu Zhiyuan Li Gyungho Lee[†]

Department of Computer Science
University of Minnesota
200 Union Street S.E.
Minneapolis, MN 55455
{gu,li}@cs.umn.edu

[†]Electrical Engineering, Division of Engineering
University of Texas - San Antonio
6900 North Loop 1604 West
San Antonio, Texas 78249-0665
glee@voyager1.eng.utsa.edu

Abstract

Array data flow analysis is known to be crucial to the success of array privatization, one of the most important techniques for program parallelization. It is clear that array data flow analysis should be performed interprocedurally and symbolically, and that 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 documents our experience with building a highly efficient array data flow analyzer which is based on *guarded array regions* and which runs faster, by one or two orders of magnitude, than other similarly powerful tools.

1 Introduction

By now it is well recognized that array data flow analysis is crucial for array privatization, one of the most important techniques for program parallelization. It is also clear that, in order to be effective, array data flow analysis must be performed interprocedurally and symbolically. Moreover, the predicates represented by IF conditions often need to be analyzed. The problem of array data flow analysis has been examined in various limited forms in the past (*c.f.* Related Works below). Recently, our group [11] and the Illinois Polaris group [24] have both proposed comprehensive frameworks for symbolic array data flow analysis which can handle IF conditions. While we perform interprocedural analysis by analyzing the call effects, the Polaris group relies on in-lining to remove the calls.

Because array data flow analysis must be performed over a large scope, handling 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. Unfortunately, the efficiency issue has not received enough attention to date. SUIF and Polaris, for example, do not emphasize compiler efficiency, and they tend to run very slowly. We believe it is important to demonstrate that array data flow analysis can be performed efficiently, and we do so in this paper by sharing our experience of building such an analyzer which runs faster, by one or two orders of magnitude, than other similarly powerful tools, e.g. the Polaris analyzer. Through this work, we wish to convince readers that sophisticated array data flow analysis can be fast enough to be used in production compilers.

In the next section, we shall first introduce the framework of our analyzer. We then present data to show its effectiveness and to compare its running time with Polaris. In Section 3, we present the main reasons for the high efficiency of our analyzer and provide supporting data. Related work will be briefly discussed in section 4. Finally, we conclude the paper.

2 Array data flow analysis: Effectiveness and efficiency

The need for an array data flow analyzer which can analyze call effects, symbolic values, and IF conditions has been well documented previously [3, 11, 17]. For readers new to the field, we will briefly explain the issues through two simple examples.

Figures 1(a)&(b) show an example from ADM in the Perfect benchmark suite. Figure 1(a) shows the simplified loop DCDTZ/40, which contains a call to subroutine CPADE shown in Figure 1(b). DO loop 40 can be parallelized if array HELP is privatizable. In order to prove HELP as privatizable, the compiler needs to establish that the definitions of HELP cover its uses both in DO 30 and in routine CPADE. This can be done by comparing the symbolic upper bounds of DO 10 and DO 30 and by analyzing the call effects of routine CPADE. In addition, the IF condition in CPADE must be taken into account. Otherwise, array element HELP(0) will be mistaken as upwards exposed to the exterior of DO 40, causing unnecessary data copy-in.

IF conditions may even affect the privatizability of arrays. Figure 1(c) illustrates the simplified loop FIL-

[1]This work is sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. This work is also supported in part by a National Science Foundation CAREER Award, Grant CCR-950254, a National Science Foundation Academic Research Instrumentation Grant, CDA 9414015 and a funding, as a part of DICE project, from Samsung Electronics. We also thank Trung N. Nguyen and Guohua Jin for their important contributions to Panorama.

```
      DO 40 J=1,NY
        DO 10 K=1,NZ
          HELP(K)=.
          ......
10:     CONTINUE
        ......
        CALL CPADE(HELP,NZ,...)
        DO 30 K=1,NZ
          . = HELP(K)
          ......
30:     CONTINUE
40:   CONTINUE
```

(a)

```
SUBROUTINE CPADE(HELP, NS,...)
      ......
      N1 = NS - 1        ......
      DO 10 J=1, N1
        ......
        IF (J.EQ.1) THEN
          . = HELP(J+1)
        ELSE
          . = HELP(J-1)
        ENDIF
10:   CONTINUE
    ......
END
```

(b)

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

(c)

Figure 1: Examples of Privatizable Arrays.

ERX/DO15 from the ARC2D program. Consider loop I. Without analyzing the effect of the IF condition, the use of array element A(jmax) in each iteration of loop I would be considered as potentially upwards exposed to the write references in the previous iterations. This forces the compiler to assume a loop-carried flow dependence which prevents array A from being privatized for loop I. However, by examining this IF condition, the compiler would recognize that the array element A(jmax) takes a value defined either outside loop I or within the same iteration in which A(jmax) is used. This assures that A(jmax) does not cause a loop-carried flow dependence. Moreover, it is easy to see that the use of A(jlow:jup) in one iteration is not upwards exposed to the previous iterations either. Hence, A is privatizable and loop I is a parallel loop.

To handle these issues simultaneously, we have designed a framework which is described below.

2.1 Guarded array regions

Our analysis is based on two basic sets which describe array references, the upwards exposed use set (UE set) and the modification set (MOD set). Given a program segment, its UE set is the set of all the array elements which are used within the segment but whose values are written outside the segment. The MOD set is the set of array elements written within the program segment.

Our basic unit of array reference representation is a *regular array region*, which is also called a *bounded regular section* [13]. It is a reduced form of the original regular sections proposed by Callahan and Kennedy [4]. (For simplicity, we refer to bounded regular sections as regular sections where this will cause no confusion.) On the other hand, we extend the original regular sections in the following ways to meet our needs in representing UE and MOD sets. First, since references to an array often cannot be easily represented by a single regular section, we have used a list of regular sections for the representation without sacrificing exactness. In addition, we annotate regular sections with predicates which affect the array references, resulting in a *guarded array region* (GAR). We presented these concepts in a previous paper [11], and have since made some important improvements.

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, 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. □

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 $(l : unknown : s)$.

Our previous work [11] uses a list of GAR's for both a MOD set and a UE set. Since then, we have improved the representation of a UE set by using a GAR with a difference list (GARWD). Its contribution to compiler efficiency will be discussed in section 3.

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*> }. □

Figure 2 is an example showing the use of 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

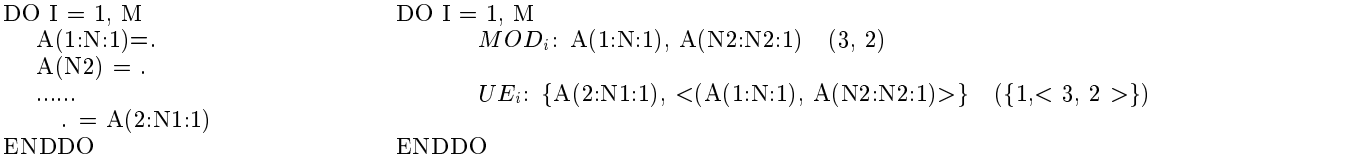

Figure 2: Example of GARWD's

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 2, which makes intersection a simple integer operation.

2.2 Operations on GAR's

Set operations for GAR's are based on set operations for regular array regions as well as on logical operations for predicates. The general formula has been given in our previous work [11]. Here, we emphasize our new improvements. Given two GAR's, $T_1 = [P_1, R_1]$ and $T_2 = [P_2, R_2]$, we describe the set operations below:

- $T_1 \cap T_2 = [P_1 \wedge P_2, R_1 \cap R_2]$
  The intersection operation is needed in array-region-based data dependence tests, array privatizability tests, and in the simplification of array regions.
- $T_1 \cup T_2$
  Two cases of union operations are the most frequent:
  - If $P_1 = P_2$, the union becomes $[P_1, R_1 \cup R_2]$
  - If $R_1 = R_2$, the result is $[P_1 \vee P_2, R_1]$

  Since these regions are symbolic, care must be taken that union operations will not create invalid regions. For example, given $R_1 = [m : p : 1]$ and $R_2 = [p + 1 : n : 1]$, the union result $R_1 \cup R_2 = [m : n : 1]$ is valid if and only if both R_1 and R_2 are valid.
- $T_1 - T_2 = [P_1 \wedge P_2, R_1 - R_2] \cup [P_1 \wedge \overline{P_2}, R_1]$
  As described in our previous paper [11], the actual result of $R_1 - R_2$ may be multiple regular array regions, making the actual result of $T_1 - T_2$ potentially complex. However, as Figure 2 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 data dependence tests or array privatizability tests. When the difference is not yet solved by the above formula, it is represented by a GARWD.

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.

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.

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.

2.3 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 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.

Figure 3 shows how the MOD and UE sets are summarized for three basic components of flow graphs, where MOD_IN(p) and UE_IN(p) denote MOD and UE sets at the location p in the flow graphs, 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 3. Figure 3 shows that, when summary sets are propagated to IF branches, IF conditions are put into the guards by applying function $padd()$ to the summary sets whenever necessary. The whole summary process is quite straightforward, except that the computation of UE sets for loops needs further analysis to support *summary expansion*.

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:

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

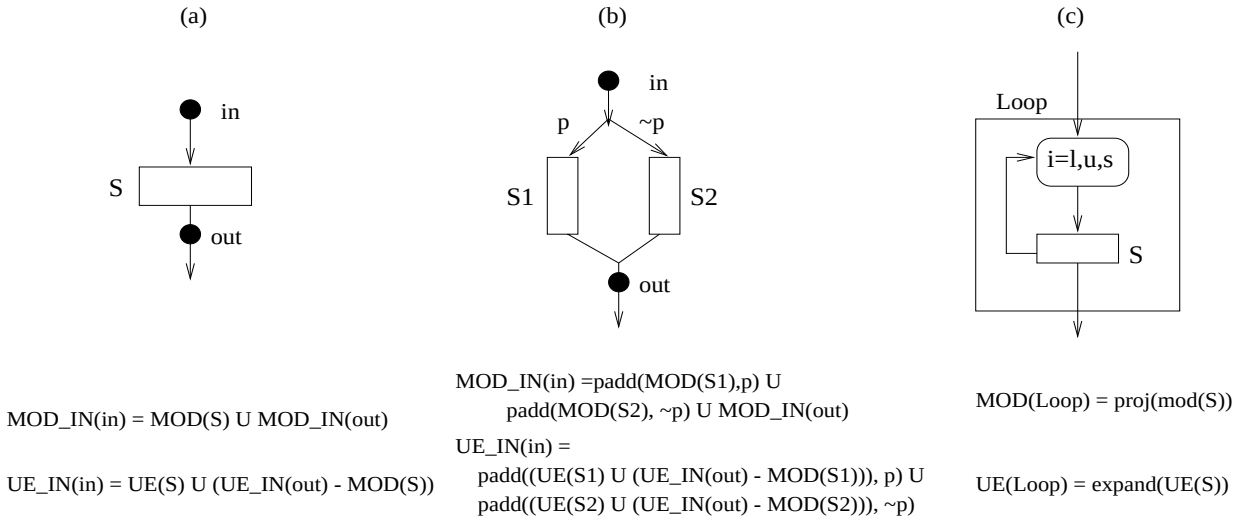

Figure 3: Computing Summary Sets for Basic Control Flow Components

The Σ summation above is also called a *projection*, denoted by $proj()$ in Figure 3, which is used to eliminate i from the summary sets. The UE calculation given above, denoted by $expand()$ in Figure 3, can be described in 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 computes the projection of Step 1's results. 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. 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 $(l'' : u'' : s'')$.

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 $UE_i = \{A(I+n), <>\}$ and $MOD_i = A(I+m)$. We can formulate

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

To be more specific, 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 $UE_{i'}$, $i\prime \in (2 : 2)$.

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.

2.4 Effectiveness and efficiency

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 [18]. 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: Privatizable Arrays and Privatization Techniques in Loops

Program	Routine /Loop	SA	PA	IA	Privatizable Arrays
ADM	dcdtz/40	Yes	Yes	Yes	HELP, DKS, CONV, HELPA, AN, BN, CN
	dtdtz/40	Yes	Yes	Yes	UNT, DTM, DKS, CONV, FORC, AN, BN, CN, HELPA
	dudtz/40	Yes	Yes	Yes	DUM, DKS, CONV, FORC, AN, BN, CN, HELPA
	dvdtz/40	Yes	Yes	Yes	DVM, DKS, CONV, FORC, AN, BN, CN, HELPA
	dkzmh/30	Yes	No	Yes	U, T
	dkzmh/60	Yes	No	Yes	U, T
	wcont/40	Yes	No	Yes	HELP, AN, BN, CN, HELPA
TRACK	nlfilt/300	No	No	Yes	XSD, P1, PP1, P2, PP2, P, PP
MDG	interf/1000	Yes	Yes	Yes	RS, XL, YL, ZL, FF, GG
	poteng/2000	No	No	Yes	XL, YL, ZL, RS, RL
TRFD	olda/100	Yes	No	No	XRSIQ, XIJ
	olda/300	Yes	No	No	XIJKS, XKL
OCEAN	ocean/270	Yes	No	Yes	CWORK
	ocean/480	Yes	No	Yes	CWORK, CWORK2
	ocean/500	Yes	No	Yes	CWORK
ARC2D	filerx/15	Yes	Yes	Yes	WORK
	filery/39	Yes	No	No	WORK
	stepfx/300	Yes	No	Yes	WORK, LDA, LDB, LDS, LUS
	stepfy/420	Yes	No	Yes	WORK
QCD	measur/3	Yes	Yes	Yes	COORD, DOWN, TOP, UP, BOTTOM, STEMP
BDNA	actfor/240	No	Yes	No	IND, XDT, YDT, ZDT
FLO52	step/20	Yes	No	No	QSI, CSI, QSJ, CSJ
MG3D	migrat/200	Yes	No	Yes	PD1, PM1, CDPM1, CDPP1
SPEC77	gwater/1000	No	No	Yes	PLN, PS, QF, DLAM, DLAMF, DPHIF, B, F, G, TAU
	gloop/1000	No	No	Yes	PLN, PS, QF, DLAM, DLAMF, DPHI, DPHIF, DER, U1, U2, V1, V2, T1, T2, Q1, Q2, CG, TAU, A, B, RTG, F, G
Total		80%	32%	80%	

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

Table 1 shows the Fortran loops in the Perfect benchmark suite which can be parallelized after our array data flow analysis and array privatization and after necessary transformations such as induction variable substitution, parallel reduction, and even synchronization placement, whose discussions are omitted in this paper. Only the Illinois Polaris tool is known to demonstrate equal power [8]. This table also marks which loops require symbolic analysis, predicate analysis and interprocedural analysis, and privatizable arrays, respectively.

Table 2 and Table 3 compare the efficiency of our analyzer with that of Polaris. Both Panorama and Polaris are compiled by the GNU gcc/g++ compiler. Two versions are produced, one without gcc's optimization (Table 2) and the other with the -O optimization (Table 3). Following a Polaris term, we call these two versions *unoptimized* and *optimized*, respectively. Table 2 also breaks down the timing for different phases of Panorama. "Parsing time" is the time to parse the program once, although Panorama currently parses a program three times. 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 4 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 40%), the total execution time is small (38 seconds maximum). The column marked "Polaris" in Table 2 shows the time spent by Polaris up to the point of array privatization and data dependence tests. Time spent after this point is not counted. Our analyzer is shown to be faster by a few orders of magnitude. (We did not compare our analyzer with SUIF, because SUIF's current public version does not perform array data flow analysis.)

Table 3 shows that the speed increase from the unoptimized version to the optimized one is more significant for Polaris than for Panorama. 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 29MB unoptimized and 14MB optimized, while Panorama, written in C, has an executable image of 3MB unoptimized and 1.1MB optimized. Even with a memory size as large as 1GB, Panorama is still faster than Polaris by one or two orders of magnitude. We believe that several design choices contribute to the efficiency of Panorama. In the next section, we present some of these choices made in Panorama.

Table 2: Execution Time (in seconds) Distribution[1]

Program	Parsing	HSG & DOALL Checking	Array Summary	Code Generation	Total	Polaris[2]
ADM	3.62	9.82	21.36	3.55	38.36	6319
QCD	1.44	3.22	11.55	1.46	17.67	
MDG	0.86	1.72	2.38	0.82	5.78	1149
TRACK	1.78	3.07	6.01	1.62	12.48	1012
BDNA	2.92	6.80	4.85	2.88	17.45	2387
OCEAN	2.41	10.31	5.00	1.88	19.56	5071
MG3D	2.27	5.71	8.33	2.16	18.47	
ARC2D	2.43	5.16	27.72	2.26	37.56	1074
FLO52	1.53	4.09	0.53	1.64	7.80	814
TRFD	0.40	0.49	0.50	0.22	1.61	1082
SPEC77	2.80	7.52	7.02	2.82	20.16	5339

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

2: Polaris is a parallelizing compiler developed at the CSRD of the University of Illinois. The timing of Polaris is measured without the passes after array privatization and dependence tests. Some programs cannot be measured because Polaris aborts or takes longer than 6 hours to execute.

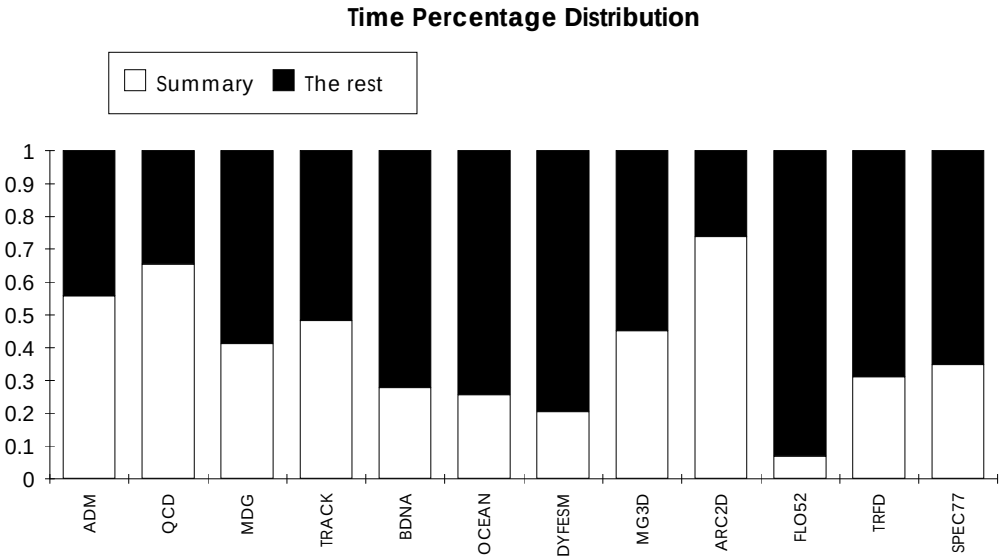

Figure 4: Time distribution for array data flow summary

3 What contributes to its efficiency?

In this section, we will discuss major reasons for the efficiency of our analyzer. 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 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. Although more work is needed to breakdown the effects of individual design choices, we believe that two design issues are particularly noteworthy, namely, the handling of predicates and the difference set operations. Next, we discuss these issues in more details.

3.1 Efficient handling of predicates

The predicate operations are expensive in general, so compilers often do not analyze 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

Table 3: Elapsed Execution Time (in seconds)

Program	SGI Power Challenge[1]		SGI Indy[2]	
	Panorama	Polaris	Panorama	Polaris
ADM	7.23	475	23.73	2030
QCD	6.54		10.15	
MDG	2.74	90.62	5.07	366
TRACK	7.01	76.89	11.05	332
BDNA	6.84	166.32	15.54	755
OCEAN	2.35	317	9.68	1660
MG3D	6.41		17.01	
ARC2D	9.18	75.87	22.58	343
FLO52	3.66	61.86	7.05	290
TRFD	1.05	55.35	1.51	285
SPEC77	4.57	355	15.49	1789

Note: Both Polaris and Panorama are compiled with gcc -O. [1]SGI Power Challenge with 1024MB memory and 196MHZ R10000 CPU. [2]SGI Indy with 134MHz MIPS R4600 CPU and 64 MB memory.

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 [21, 5]. 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 [11]. 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 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 5 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 6, 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 6. We use a unique token

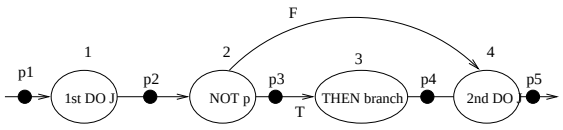

Figure 7: The HSG of the Body of the Outer Loop for Figure 1(c)

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. Columns 7 and 8 in Table 4 shows that over 90% 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.

The numbers shown in Table 4 are dependent on the strategy used to handle the predicates. Currently, we defer the checking of predicate contents until the last step, so that only a few low level 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.

3.2 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. no need to perform them). In Table 4, the total number of difference operations and the total number of reduced difference operations are illustrated in columns 5 and 6, respectively. The result shows that difference operations overall are reduced by about 17%.

Let us use the example in Figure 1(c) to further illustrate this fact. A simplified control flow graph of the body of the outer loop is shown in Figure 7. (For more information about our control flow graph, please consult our paper [11].) Suppose that each node has been summarized and that the summary results are listed below:

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

Following the description given in Section 2.3, 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. $MOD\_IN(p4) = MOD(4) = \emptyset$
   $UE\_IN(p4) = UE(4) = [T,(jlow:jup)] \cup [T,(jmax)]$

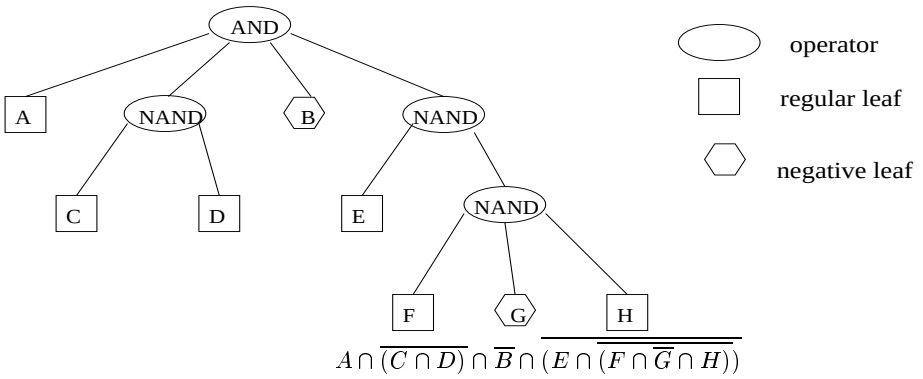

Figure 5: High level representation of predicates

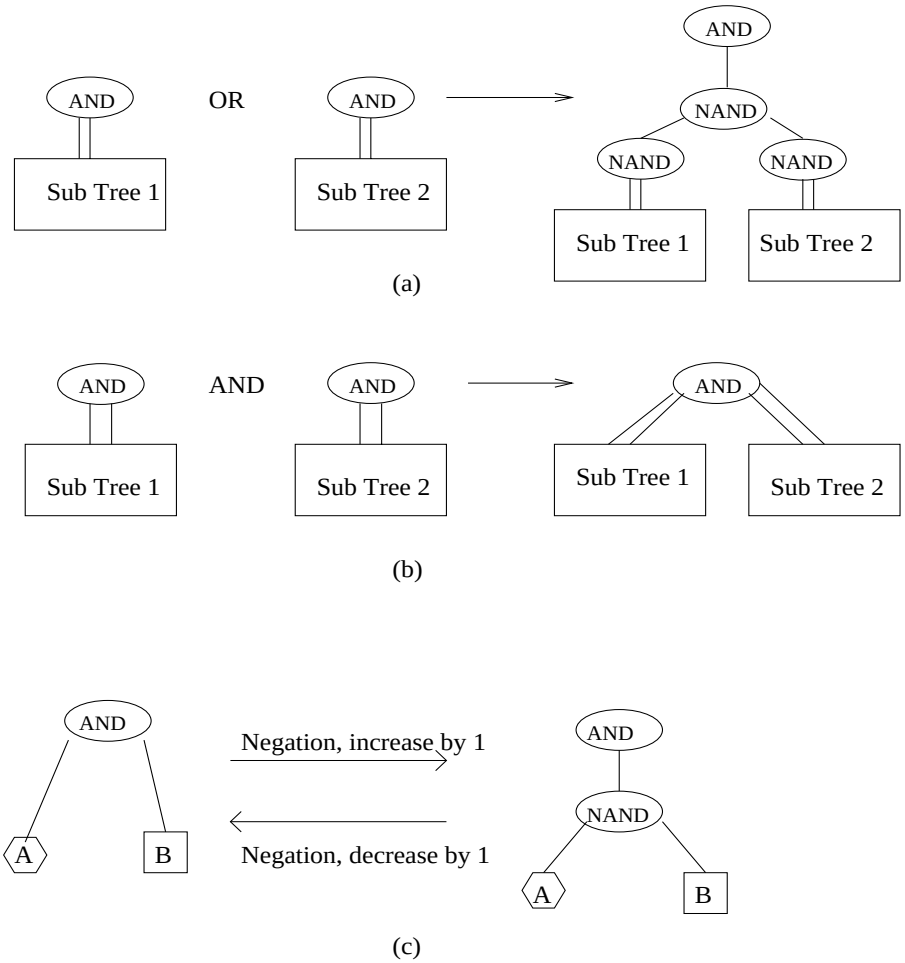

Figure 6: Predicate operations

Table 4: Measurement of key parameters

Program	# Array Summarized	Ave # GAR's	Ave # GARWD's	Difference Ops		# Predicate Ops	
				Total	# Diff Ops Reduced	Total	# High Level Ops
ADM	630	1.82	2.46	630	362	5001	4740
QCD	326	1.93	6.33	942	21	15711	14173
MDG	103	1.79	3.11	177	70	98	98
TRACK	115	1.45	1.46	241	6	938	890
BDNA	284	1.46	1.93	238	83	638	609
OCEAN	18	2.11	1.33	15	2	0	0
MG3D	189	3.25	9.07	132	51	443	139
ARC2D	531	4.93	2.25	1753	137	7883	7883
FLO52	30	1.75	2.75	9	0	84	84
TRFD	57	2.12	2.00	18	4	97	35
SPEC77	398	1.88	4.43	153	8	383	219
Total	2681	2.23	3.37	4308	744	31276	28870

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 have 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 by one mod each from $MOD_{<i}$ and UE_i. 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.

3.3 Efficient summary sets

Our GAR's and GARWD's are based on the regular regions, which represent summary sets more efficiently than the convex regions used in several other works. Slightly differently from the regular sections originally proposed by Callahan and Kennedy and later enhanced by Havlak [4, 13], we use a list of GAR's or GARWD's to keep a precise summary set. Keeping a long list potentially can be inefficient, because the time complexity of set operations of two region lists with lengths n and m is in the order of $(n \cdot m)$. In our previous work [11], we merged two regions whenever possible by adding conditions to the guards. This treatment guarantees that no invalid regions are created due to invalid inequalities in the region limits, and it also shortens the length of a region list. However, at the same time, this treatment produces additional predicates to be handled, which we want to avoid. In our current design, we keep regions in an unmerged list unless the merged result is known to be valid without adding conditions to the guard. Fortunately, the average lengths of a MOD and a UE list are not long, as shown in Table 4 by the two columns marked "ave # GAR's" and "ave # GARWD's".

4 Related works

There are 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 [9] calculates the *source function* to indicate detailed flow information. Maydan et al. [16, 17] simplify Feautrier's method by using a Last-Write-Tree(LWT).

Duesterwald et al. [7] compute the dependence distance for each reaching definition within a loop. Pugh and Wonnacott [19] use a set of constraints to describe array data flow problems and solve them basically by the Fourier-Motzkin variable elimination. Maslov [15], as well as Pugh and Wonnacott [19], 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* [4, 13], convex regions [22, 23], 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 [10], Rosene [20], Li [14], Tu and Padua [25], Creusillet and Irigoin [6], and M. Hall et al. [12]). 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 upwards 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 not useful in some cases. Tu and Padua [25], and Creusillet and Irigoin [6] seem to use a single regular section and a single convex region, respectively. M. Hall et al. [12] 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.

5 Conclusion

This paper presents an efficient design of array data flow analysis which handles interprocedural, symbolic, and predicate analyses all together. As far as we know, this is the first time the efficiency issue has been addressed and data presented for such a powerful analysis. The efficiency is improved by several design considerations such as GARWD's and a hierarchical predicate handling scheme. Our preliminary results show that our approach is much faster than similarly powerful tools.

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] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

[19] 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.

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

[21] Jr T.E. Cheatham, 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.

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

[23] 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.

[24] 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.

[25] 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.