Array Privatization for Parallel Execution of Loops

Zhiyuan Li*
Department of Computer Science
University of Minnesota
200 Union St. SE, Minneapolis, Minnesota 55455
li@cs.umn.edu

Abstract

In recent experiments, array privatization played a critical role in successful parallelization of several real programs. This paper presents compiler algorithms for the program analysis for this transformation. The paper also addresses issues in the implementation.

1 Introduction

The diversity of parallel architectures makes it difficult to write efficient parallel programs in a machine independent language. For a long time, many researchers have pursued the goal of automatic transformation of sequential programs into parallel machine code. Unfortunately, the result has been unsatisfactory. Many transformation techniques used in existing compilers do not prove to be effective in practice [EB91], mainly because they handle relatively simple cases. On the other hand, recent experiments show significant results by solving more complex cases, using hand-performed new analyses and transformations [EHJ+91], [EHLP91]. A technique called *array privatization*, along with other techniques, improves the performance of all four programs reported in [EHLP91]. Tables 1 and 2 summarize the effect of array privatization on those programs. The fifth column of Table 2 shows the percentage of the sequential execution time (over the whole program) of each loop which is made parallel after array privatization. The percentage indicates the significance of each loop. For every program except OCEAN, no improvement would result without array privatization. We have improved other application programs through array privatization, although we have yet to compile the result data to show the effect. These programs include ADM and SPEC77 in the Perfect program suite [PER89] and WHAMS3D [BT82], a program for structural analysis.

In this paper, we discuss compiler algorithms for automatic *array privatization*. This technique may be viewed as an extension of *scalar privatization* which has been used in parallelizing compilers [ABC+88], [CF87]. We may call both techniques *variable privatization*. If a variable is modified in different iterations of a loop, writing conflicts result when the iterations are executed by multiple processors. Variable privatization removes such conflicts by allocating in each processor a private storage for the offending variable. Parallelism in the program is increased. Data access time may also be reduced, since privatized variables can be allocated to local memories. Of course, the effectiveness of privatization is limited by the memory size.

Table 1. Speedups before and after hand transformations on the Alliant FX/80 (8 vector processors).

Code	Before	After	Array Priv.
MDG	1.1	5.5	Yes
OCEAN	1.42	8.3	Yes
TRACK	0.90	5.1	Yes
TRFD	2.36	13.2	Yes

Table 2. Loops in which arrays are privatized (speedup is over seq. time).

Prog	Loop Label	Routine	Loop Spdup	% of Seq
TRACK	300	nlfilt	5.2	40%
	300	fptrak	6.0	9%
	400	extend	7.0	34%
MDG	1000	interf	6.0	90%
	2000	poteng	5.2	8%
TRFD	100	olda	16.4	69%
	300	olda	12.3	29%
OCEAN	270	ocean	8.0	3%
	480	ocean	6.1	4%
	500	ocean	6.5	3%

Privatization is not always valid. If the value of a

*This work is supported by the Graduate Collage of University of Minnesota.

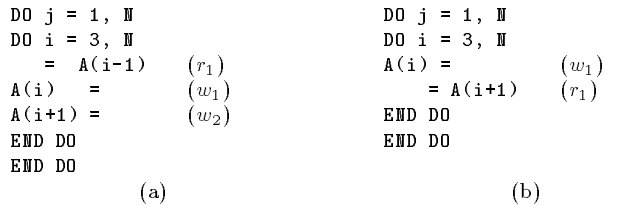

Figure 1: Examples of SDDs and NSDDs

variable is defined in one iteration and used in another, then the variable cannot be privatized. Privatizable scalars can easily be identified. If any scalar definition reaches the end of the loop body and any use of the same scalar is exposed to the beginning of the loop body, then the definition will certainly reach the use from one iteration to another, which prohibits privatization. However, such a criterion does not apply to array references. In the simple example in Figure 1(a), w_1 and w_2 reach the end of the loop body and r_1 is exposed to the beginning of the loop body. However, r_1 does not read any value generated in previous iterations of the outer loop. Array A, therefore, can be privatized in the outer loop. In contrast, A cannot be privatized in the outer loop in Figure 1(b), because its values are carried from one iteration to another. From the examples, it is clear that new criteria are needed for array privatization.

We organize this paper as follows. In Section 2, we discuss the basic idea and various aspects of array privatization. We also review the related work. In Section 3, we present algorithms to determine array privatizability, which is the key to the technique. In Section 4, we present an algorithm for live array analysis which is used to determine whether *copy-out* is necessary. We summarize in Section 5.

2 Overview and Related Work

Two types of writing conflicts may exist between different iterations of a loop. The first type involves a single definition which overwrites the same array elements in different iterations. The second type involves two different definitions. Privatization is suitable for the first type, while *renaming* is suitable for the second. In Figure 1(a), the writing conflicts between w_1 and w_2 can be removed by renaming, i.e., by letting w_2 write to a new array. This paper covers privatization only.

The problem of privatization may be divided into three issues. The first is the privatizability of an array in a loop. The second is the need to *copy out* the last defined values. The third is the storage allocation for privatized arrays. We describe these issues separately in the following.

2.1 Array privatizability

First, we define the object of privatization.

Definition If a definition of a variable overwrites the same array element(s) in different iterations of a loop L, we call this definition a *self dependent definition* (SDD) in loop L.

In Figure 1(a), both w_1 and w_2 are SDDs in the outer loop, because in every iteration, w_1 writes the same elements, A(3:N), and w_2 writes the same elements, A(4:N+1). However, w_1 and w_2 are not SDDs in the inner loop, because they write different elements in different iterations. Similarly, in Figure 1(b), w_1 is an SDD in the outer loop, but not in the inner loop. SDDs can be recognized by the well-understood analysis of *output dependences* [Ban88].

Definition An SDD w is *privatizable* in loop L *iff* the value generated by w is not used across the iterations of L.

In Figure 1(a), both w_1 and w_2 are privatizable in the outer loop. In Figure 1(b), w_1 is not privatizable. These facts will be explained as our discussion proceeds.

In a DO loop, we can divide the definitions of an array into three categories: the non-self dependent definitions (NSDDs), privatizable definitions, and nonprivatizable definitions. We privatize an array in a DO loop only if the following criteria are both met.

Criterion A The SDDs of the array are all privatizable in the loop.

Criterion B The SDDs do not overlap the NSDDs in the loop. [1]

Breaking the two criteria would make the compiler analysis substantially more complicated. To break Criterion A, the compiler may privatize the privatizable definitions while leave others unchanged. However, the compiler will have to identify which definitions are privatizable and which are not. To break Criterion B, the compiler must be able to handle the situation in which both a privatizable definition and an NSDD may reach an array use. If either of the criteria is broken, the compiler has to analyze array reaching definitions fully and exactly, which is extremely time-consuming. On the other hand, it seems unlikely in practice that a better performance could result by breaking the two criteria. In the experiment reported in [EHLP91], the privatized arrays meet both the above criteria.

To check whether an array A meets Criterion B in a DO loop, we intersect the set of the A elements defined by SDDs and the set of the A elements defined by NSDDs. This is a typical data dependence test problem

[1] They may overlap in an outer loop.

and has been studied extensively [Ban88]. Hence, we need to examine Criterion A only.

Once the privatizable arrays are identified, the compiler can prune the data dependence graph. All *loop carried* dependences whose sources are SDDs can be eliminated. Of course, other transformations such as scalar privatization can also contribute to the graph pruning. Based on the new dependence graph, the compiler decides which loops should be executed in parallel. Variables are privatized in parallel loops only.

2.2 The copy-out issue

If any self dependent definitions (SDDs) in a loop reach the array uses outside the loop, then the last values written by the SDDs must be copied out. When a processor executes a particular iteration, it needs to determine which array elements should be copied out. Figure 2 shows simple examples. In each example, A is privatizable in the outer loop according to Criteria A and B. Copy-out is necessary in (a), but is unnecessary in (b) because the SDDs are *killed* in the following loop by an assignment to A(k). For program TRFD (cf. Table 2), two privatized arrays are copied out from each of the two listed loops. For other programs in Table 2, copy-out is unnecessary either because the privatized arrays are not referenced outside the loops, or because the definitions are killed before reaching any uses.

Definition The *last defining iteration* (LDI) of an array element is the iteration in which the element is last defined in the loop.

IF statements in a loop tend to make it unclear which iteration is the last defining iteration for a particular arary element. In this case, a processor cannot determine independently, i.e. without communicating with others, whether the current iteration is the LDI. We say that, in this case, the LDI is *independently undeterminable*. The LDIs in Figures 2(a) and 2(b) are independently, and easily, determinable. For every A element, the LDI is $i = N$. LDIs of the privatized arrays in program TRFD (cf. Table 2) are also independently, and easily, determinable. In Figure 2(c), however, the LDIs are independently undeterminable. In this case, we do not privatize the array. In our experience so far, the LDIs are independently determinable whenever we need to copy out. However, it is unclear whether this is generally true in practice.

To summarize, we handle the copy-out issue as follows. First, the compiler performs a fast *live array* analysis to see whether the SDDs in a loop reach any outside use. If no SDDs of array A reach any outside use, then no copy-out is necessary for A and we mark A as a *candidate* for privatization. Otherwise, we examine whether the LDIs are independently undeterminable. If they are, we do not privatize A. If they are not, then we mark A as a candidate for privatization. The copy-out code will be described in the next subsection. The algorithms for the live array analysis will be briefly discussed in Section 4. Reference [Li92] discusses algorithms for determining LDIs.

```
DO i = 1, N
DO j = 1, M
A(j)   =
   =  A(j)
END DO
END DO

DO k = 2, M-1
   =  A(k)
END DO
```

(a)

```
DO i = 1, N
DO j = 1, M
   A(j)  =
      =  A(j)
END DO
END DO

DO k = 1, M
A(k)   =
   = A(k)
END DO
```

(b)

```
DO i = 1, N
DO j = 1, N
  IF () THEN
  A(j) =
     = A(j)
  END IF
END DO
END DO
```

(c)

Figure 2: Copy-out and LDIs

2.3 Storage allocation

The storages are allocated as follows. For each parallel loop, the compiler checks to see whether it immediately contains any self dependent definitions (SDDs) of a candidate for privatization. If it does, then in each processor that participates in executing the loop, the compiler allocates a private storage for each of such candidates. This materializes the array privatization. (The storage will be reclaimed as soon as the loop is completed.) Note that sometimes the SDDs in a loop modify a part of an array only. In these cases, it suffices to allocate a private storage just for that part of the array.

In all programs listed in Table 2, the privatized arrays are privatizable at one loop level only. Given a *perfect nest* of loops that have no intervening statements between the loop headers or the loop ends, it always suffices to privatize the array in the innermost loop in which the array is privatizable. However, if a loop nest is imperfect and an array is privatizable in more than one loop, then the array should be privatized in more than one loop. It is clear that a privatizable SDD should write into the private storage allocated to the innermost enclosing loop in which the array is privatized. We call this loop *the home loop* or simply the *home* of the SDD. If the array of an SDD is not privatized in any enclosing loop, then the SDD should write into the storage for the original array. In this case, we say the *home* of the SDD is the whole scope of the original array, say, a subroutine. For an array definition which is not an SDD in any loop, the home is also the scope of the original array. The hierarchy among the

homes for the same array can be represented by a tree whose root is the scope of the original array. In the programs in Table 2, all privatized SDDs have one home only. Nonetheless, in program SPEC77 in the Perfect code suite [PER89], some SDDs have two homes, one of which being the scope of the original array.

For each array use, the compiler must determine the storage from which it reads. This depends on which array elements are used. The compiler locates the nearest home that satisfies the following: (1) the home contains the array use; and (2) the used element is defined by SDDs in that home. In other words, the used element should not be *upward exposed* (defined in Section 3) in the home.

Copy-out, when necessary, is performed as follows. When a processor executes a loop iteration, it checks to see whether the iteration is the last defining iteration of some privatized array elements that must be copied out. The processor writes these elements into the storage allocated to the home, H, at the next upper level. If H is a DO loop and there exist statements outside H which use the copied elements, then the processor which executes the last iteration of H is responsible for copying the elements all the way up to the parent of H. This process continues until all statements that use the copied elements are served. Copy-out in such "relay" style simplifies the analysis of reaching definitions. In our experience so far, e.g. in program TRFD (Table 2), only one step of copy-out is necessary.

We use examples (Figure 3) to show the code before and after array privatization. To make them easy to understand, we present the examples in a self-evident pseudo parallel language. The keyword `DOALL` indicates a parallel loop and the keyword `private` indicates the new name for a privatized array. In example (a), the inner loop is the only home loop. The array use reads most elements generated within the loop. But it also reads an element from outside the loops. Here, we simply copy in that element A(1). In example (b), there are two home loops. An array element is copied from one home to the other.

Implementation of the mechanism described in this subsection is quite straightforward. Therefore, we do not provide algorithms here to formalize the mechanism. Analysis of array privatizability, on the other hand, is essential but not straightforward. Hence, we present its algorithms in Section 3.

2.4 Related work

Literature on data dependence analysis abounds and is well-known. Data dependence analysis, in the commonly understood terms, are concerned about intersection of array regions, which does not support array

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

(a)

```
DO i = 1, N                 DOALL i = 1, N
DO j = 1, N                   private A1(1:N)
DO k = 1, N                 DOALL j = 1, N
A(k)   =                      private A2(1:N)
   =  A(k)                  DO k = 1, N
END DO                      A2(k) =
END DO                            = A2(k)
                            END DO
                            IF(j=N) THEN
                            A1(1) = A2(1)     copy-out
                            END DOALL
DO h = 2, N                 DO h = 2, N
A(h)   =                    A1(h) =
   =  A(h-1)                    =  A1(h-1)
END DO                      END DO
END DO                      END DOALL
```

(b)

Figure 3: Examples of array privatization

privatization. Nonetheless, we have shown that the results in data dependence analysis are useful for identification of self dependent definitions and for verification of Criterion B. Our work has been inspired by recent related work. Feautrier studies the technique of *array expansion* [Fea88]. His method expands arrays to higher dimensions to allow a program transformed into a single assignment form. He considers programs which have DO loops, assignment statements and conditional assignment statements. Gross and Steenkiste devise a dataflow analysis on arrays [GS90]. Their algorithm examines whether a definition reaches a use in the same loop iteration and whether it reaches the end of the loop body.

Our work focuses on the information required to determine array privatizability. The essential information is whether an array reference can possibly read a value from previous iterations, suppose the loop is executed in sequential. This information is not provided by either the traditional data dependence analysis or recent related works. To make it more general, we consider arbitrary flow graphs including cases in which a DO loop has multiple exits. Interestingly, in our scheme, if

a program's flow graph is reducible, we do not need to propagate information over any cycles.

3 Determination of Array Privatizability

3.1 An assumption on IF conditions

Most data flow analyses are insensitive to the exact content of an IF condition. Unfortunately, for subroutine INTERF of MDG and the main program of OCEAN (cf. Table 2), array privatizability cannot be recognized without considering the exact contents of IF conditions. In the case of MDG, the conditions are in the form of "if (LOC < 1 or LOC+NW-1 > MEMSIZE or NW < 1)", which involves scalars only. In the case of OCEAN, the conditions are in the form of "if (RS(k) > CUT2)", which involves arrays as well. Compiler mechanisms that deal with scalar relations do exist [LT88], [TIF86]. But mechanisms dealing with arrays are yet to be devised. Nonetheless, to separate two different issues, we do not examine the contents of IF conditions in this paper. As a result, we must make the assumption that one branch decision will not affect another.

The above assumption makes our computation of the *upward exposed* array elements (defined below) conservative. It also implies that nonprivatizable definitions exist in loop L if and only if an array element is defined by self dependent definitions (SDDs) in one iteration and is upward exposed in a later iteration, which is a conservative statement.

3.2 Upward exposed array elements

Definition If an array element is always defined before used in iteration i, we say the element is *covered* in iteration i. Otherwise, we say the element is *upward exposed* in iteration i.

In order to determine whether nonprivatizable definitions exist, we introduce the following notations. Consider a nest of DO loops, L_1, L_2, ..., L_n, where L_1 is the outermost, L_2 is nested in L_1, and so on. Suppose the loop index variables are I_1, I_2, ..., I_n, respectively. (Without loss of generality, we assume index steps of 1.) We represent an iteration of L_d, $1 \le d \le n$, by an index vector $\langle i_1, i_2, \ldots, i_d \rangle$ such that $I_1 = i_1$, $I_2 = i_2$, and $I_d = i_d$. For a given array A, let $UE_{\langle i_1, i_2, \ldots, i_d \rangle}$ denote the set of all upward exposed elements in iteration $\langle i_1, i_2, \ldots, i_d \rangle$ and let $D_{\langle i_1, i_2, \ldots, i_d \rangle}$ denote the set of all elements which *may* be written in $\langle i_1, i_2, \ldots, i_d \rangle$ by SDDs. The following formalizes our condition for the existence of nonprivatizable definitions.

Claim 1 *Loop L_d has nonprivatizable definitions of A* iff, *for some iteration $\langle i_1, i_2, \ldots, i_d \rangle$, the intersection of $UE_{\langle i_1, i_2, \ldots, i_d \rangle}$ and $\bigcup_{l_d \le k \le i_d - 1} D_{\langle i_1, i_2, \ldots, k \rangle}$ is nonempty, where l_d is the lower bound of L_d and $i_d > l_d$.*

Consider the outer loops in Figure 1. In (a), A(2) is the upward exposed element in every iteration, but A(2) is not defined in any iterations, because $D_{\langle j \rangle}$ is A(3:N+1). Therefore, the definitions of A are privatizable in (a). In (b), $D_{\langle j \rangle}$ is A(3:N). A(4:N) are both upward exposed and defined in every iteration. Therefore, w_1 is not privatizable.

The *cover set* defined below is instrumental in computing the set of upward exposed elements. Suppose statement S is nested in L_1, L_2, ..., L_n. Let $S_{\boldsymbol{\alpha}}$ denote the instance of S in iteration $\boldsymbol{\alpha} = \langle i_1, i_2, \ldots, i_n \rangle$ and $S_{\langle i_1, i_2, \ldots, i_d \rangle}$ denote *all* instances of S in iteration $\langle i_1, i_2, \ldots, i_d \rangle$. We may also write $S_{\langle i_1, i_2, \ldots, i_d \rangle}$ as $S_{\boldsymbol{\alpha}|_d}$, where $\boldsymbol{\alpha}|_d$ stands for the first d elements of $\boldsymbol{\alpha}$.

Definition For array A, the *cover set* of $S_{\boldsymbol{\alpha}}$ *at loop level d* is the set of the A elements that are guaranteed to be defined *within* iteration $\boldsymbol{\alpha}|_d$ before the execution of $S_{\boldsymbol{\alpha}}$. $C^S_{\boldsymbol{\alpha}}|_d$ denotes this cover set.

Take the examples in Figure 1, where we suppose S is the statement that issues r_1 and $\boldsymbol{\alpha}$ stands for $\langle j, i \rangle$. In (a) and (b), the cover set of $S_{\boldsymbol{\alpha}}$ at loop level 1 is A(3:i). Note that a cover set, like other types of sets in this paper, often has loop index variables as its parameters. If we assign different values to the index variables, we obtain different sets of array elements. Set representation is briefly discussed in [Li92].

Let $U^S_{\boldsymbol{\alpha}}$ denote the array elements used by $S_{\boldsymbol{\alpha}}$. In iteration $\boldsymbol{\alpha}|_d$, the set of the upward exposed elements used by $S_{\boldsymbol{\alpha}}$ are clearly $U^S_{\boldsymbol{\alpha}} - C^S_{\boldsymbol{\alpha}}|_d$. The set of the upward exposed elements used by all instances of S in iteration $\langle i_1, i_2, \ldots, i_d \rangle$ is the union of $U^S_{\boldsymbol{\alpha}} - C^S_{\boldsymbol{\alpha}}|_d$ over all $\boldsymbol{\alpha}$ such that $\boldsymbol{\alpha}|_d = \langle i_1, i_2, \ldots, i_d \rangle$. Finally, as claimed below, we can determine the set of the upward exposed elements in iteration $\langle i_1, i_2, \ldots, i_d \rangle$.

Claim 2 *$UE_{\langle i_1, i_2, \ldots, i_d \rangle}$ equals to the union of $U^S_{\boldsymbol{\alpha}} - C^S_{\boldsymbol{\alpha}}|_d$ over all $\boldsymbol{\alpha}$ such that $\boldsymbol{\alpha}|_d = \langle i_1, i_2, \ldots, i_d \rangle$ and over all S that make use of array A in loop L_d.*

We illustrate this claim through Figure 1. Let S be the statement which issues r_1 and let $\boldsymbol{\alpha}$ be $\langle j, i \rangle$.

(a) $U^S_{\boldsymbol{\alpha}}|_1$ is A(i-1). $U^S_{\boldsymbol{\alpha}} - C^S_{\boldsymbol{\alpha}}|_1$ is A(i-1) for $i = 3$ and is empty for other i. Hence, $UE_{\langle j \rangle}$ is A(2).

(b) $U^S_{\boldsymbol{\alpha}}$ is A(i+1). $C^S_{\boldsymbol{\alpha}}|_1$ is A(3:i). Hence, $U^S_{\boldsymbol{\alpha}} - C^S_{\boldsymbol{\alpha}}|_1$ is A(i+1). $UE_{\langle j \rangle}$ is $\{\text{A(i+1)} \mid 3 \le i \le N\}$ = A(4:N+1).

3.3 Computing the cover sets

In this subsection, we discuss how the control flows affect the cover sets. We first consider DO loops that have no exits due to GOTO.

We build the control flow graphs in a hierarchical fashion. Consider again the DO loops L_1, L_2, ..., L_n. We assume that L_d may contain DO loops in addition to L_{d+1}, $d = 1, 2, \ldots, n-1$. For the loop body of L_d, we construct a *condensed* control flow graph, G_d. Each DO loop, including L_{d+1}, that is immediately nested in L_d is represented by a *condensed* node in G_d. We denote the condensed nodes by $L^{(1)}$, $L^{(2)}$, ..., $L^{(m)}$. The remaining statements in L_d are grouped into basic blocks in the same way as in the conventional control flow graphs. Each basic block is represented by a regular node in G_d. A condensed node is different from an *interval* [All70], [Coc70] in that only an indexed DO loop may be condensed. A loop of any other type is decomposed into basic blocks and is represented by a number of regular nodes.

It is convenient to index the instances of a basic block, B, or a DO loop, $L^{(i)}$, in L_d. Let $B_{\langle i_1, i_2, \ldots, i_d \rangle}$ denote the instances of basic block B in iteration $\langle i_1, i_2, \ldots, i_d \rangle$. [2] Similarly, let $L^{(i)}_{\langle i_1, i_2, \ldots, i_d \rangle}$ denote the instances of DO loop $L^{(i)}$ in iteration $\langle i_1, i_2, \ldots, i_d \rangle$. In the following, unless we state otherwise, we use the symbol P to denote either a basic block or a DO loop $L^{(i)}$.

Definition For a given array A, the *incoming-cover-set* of $P_{\langle i_1, i_2, \ldots, i_d \rangle}$ in G_d, denoted by $C\_IN^P_{\langle i_1, i_2, \ldots, i_d \rangle}$, is the set of the A elements guaranteed to have been defined in iteration $\langle i_1, i_2, \ldots, i_d \rangle$ before P is entered. The *outgoing-cover-set* of $P_{\langle i_1, i_2, \ldots, i_d \rangle}$ in G_d, denoted by $C\_OUT^P_{\langle i_1, i_2, \ldots, i_d \rangle}$, is the set of the A elements guaranteed to have been defined in iteration $\langle i_1, i_2, \ldots, i_d \rangle$ before the exit of P.

Definition The *must-modify* set of $P_{\langle i_1, i_2, \ldots, i_d \rangle}$, denoted by $M^P_{\langle i_1, i_2, \ldots, i_d \rangle}$, is the set of the A elements defined in $P_{\langle i_1, i_2, \ldots, i_d \rangle}$ no matter what control path is taken within $P_{\langle i_1, i_2, \ldots, i_d \rangle}$. Similarly, $M^S_{\langle i_1, i_2, \ldots, i_d \rangle}$ denotes the set of the A elements defined in $S_{\langle i_1, i_2, \ldots, i_d \rangle}$, an instance of statement S.

We now establish the basic flow equations regarding the sets defined above. First of all, we have

Eq1. $C\_OUT^P_{\langle i_1, i_2, \ldots, i_d \rangle} = C\_IN^P_{\langle i_1, i_2, \ldots, i_d \rangle} \cup M^P_{\langle i_1, i_2, \ldots, i_d \rangle}$.

Eq2. Suppose P_1, P_2, ..., P_p are the predecessors of P in G_d. $C\_IN^P_{\langle i_1, i_2, \ldots, i_d \rangle}$ equals $\bigcap_{i=1,p} C\_OUT^{P_i}_{\langle i_1, i_2, \ldots, i_d \rangle}$.

Eq3. $C\_IN^T_{\langle i_1, i_2, \ldots, i_d \rangle} = C\_OUT^T_{\langle i_1, i_2, \ldots, i_d \rangle}$ is empty, where T is the starting node in L_d.

[2] In the presence of **while** or **goto** loops, an instance defined as such may repeat at run time. This, however, does not disturb our discussion.

Now, suppose a basic block B contains s statements which modify A elements. We list these statements by their textual order: $S_0, S_1, \ldots, S_s$. By definitions, we have

Eq4. $M^B_{\langle i_1, i_2, \ldots, i_d \rangle} = \bigcup_{i=1,p} M^{S_i}_{\langle i_1, i_2, \ldots, i_d \rangle}$.

Eq5. For any statement S in B which textually precedes S_0, $C^S_{\langle i_1, i_2, \ldots, i_d \rangle}|_d = C^{S_0}_{\langle i_1, i_2, \ldots, i_d \rangle}|_d = C\_IN^B_{\langle i_1, i_2, \ldots, i_d \rangle}$.

Eq6. For any statement S between S_i and S_{i+1}, $0 \le i \le s-1$, $C^S_{\langle i_1, i_2, \ldots, i_d \rangle}|_d = C^{S_{i+1}}_{\langle i_1, i_2, \ldots, i_d \rangle}|_d = C^{S_i}_{\langle i_1, i_2, \ldots, i_d \rangle}|_d \cup M^{S_i}_{\langle i_1, i_2, \ldots, i_d \rangle}$.

Eq7. $C\_OUT^P_{\langle i_1, i_2, \ldots, i_d \rangle} = C^{S_s}_{\langle i_1, i_2, \ldots, i_d \rangle} \cup M^{S_s}_{\langle i_1, i_2, \ldots, i_d \rangle}$.

Eq8. $C\_IN^P_{\langle i_1, i_2, \ldots, i_d \rangle} \subseteq C\_OUT^P_{\langle i_1, i_2, \ldots, i_d \rangle}$.

Under the present assumption that a DO loop has one exit only, we insert a single exit node, X, in L_d. $C\_OUT^X_{\langle i_1, i_2, \ldots, i_d \rangle} = C\_IN^X_{\langle i_1, i_2, \ldots, i_d \rangle}$ is the set of the A elements that must be defined within iteration $\langle i_1, i_2, \ldots, i_d \rangle$.

Definition The *past-modify set* of loop L_d in iteration $\langle i_1, i_2, \ldots, i_d \rangle$, denoted by $\tilde{M}^{L_d}_{\langle i_1, i_2, \ldots, i_d \rangle}$, is the set of A elements defined in the iterations from $\langle i_1, i_2, \ldots, l_d \rangle$ through $\langle i_1, i_2, \ldots, i_d - 1 \rangle$. This set is empty if $i_d = l_d$.

Claim 3 *In DO loops $L_1, L_2, \ldots, L_n$, the past-modify sets are determined by the equation*

$$\tilde{M}^{L_d}_{\langle i_1, i_2, \ldots, i_d \rangle} = \bigcup_{l_d \le I_d \le i_d - 1} C\_OUT^{X_d}_{\langle i_1, i_2, \ldots, i_{d-1}, I_d \rangle}. \quad (1)$$

The must-modify sets are determined by the equation

$$M^{L_d}_{\langle i_1, i_2, \ldots, i_{d-1} \rangle} = \bigcup_{l_d \le I_d \le u_d} C\_OUT^{X_d}_{\langle i_1, i_2, \ldots, i_{d-1}, I_d \rangle}, \quad (2)$$

where u_d is the upper bound of L_d. The cover sets of statements are determined by the equation

$$\begin{aligned} C^S_{\boldsymbol{\alpha}}|_d \;=\; & C^S_{\boldsymbol{\alpha}}|_n \cup (C\_IN^{L_{d+1}}_{\langle i_1, i_2, \ldots, i_d \rangle} \cup \tilde{M}^{L_{d+1}}_{\langle i_1, i_2, \ldots, i_{d+1} \rangle}) \\ & \cup (C\_IN^{L_{d+2}}_{\langle i_1, i_2, \ldots, i_{d+1} \rangle} \cup \tilde{M}^{L_{d+2}}_{\langle i_1, i_2, \ldots, i_{d+2} \rangle}) \\ & \cup \ldots \\ & \cup (C\_IN^{L_n}_{\langle i_1, i_2, \ldots, i_{n-1} \rangle} \cup \tilde{M}^{L_n}_{\langle i_1, i_2, \ldots, i_n \rangle}). \end{aligned}$$

If a DO loop L_d has exits due to GOTO, the above equations need modification. First of all, we insert an exit node in G_d for each exit. The exit for the normal termination of L_d is denoted by $X_d^{(0)}$, while the other exits are denoted by $X_d^{(1)}$, $X_d^{(2)}$, and so on. In graph G_{d-1} at the upper level, suppose L_d is represented by a condensed node P. We make a copy of P in G_{d-1} for

every GOTO exit in L_d. We draw an edge from each predecessor of P to each copy of P. Suppose an exit jumps to a statement in L_{d-1}. That statement must be the header in a node, Q, in G_{d-1}. We draw an edge from each copy of P to the corresponding target Q. If an exit of L_d jumps outside L_{d-1}, then we treat the corresponding copy of P as an exit node in G_{d-1} and repeat the above process on the new exit node.

For GOTO exits, Claim 3 should be modified as follows. The past-modify set is computed for $X^{(0)}$ only, i.e.

$$\tilde{M}^{L_d}_{\langle i_1,i_2,\ldots,i_d\rangle} = \bigcup_{l_d \le I_d \le i_d - 1} C\_OUT^{X^{(0)}_d}_{\langle i_1,i_2,\ldots,i_{d-1},I_d\rangle}. \quad (3)$$

The must-modify set for the original P (corresponding to the normal exit) is computed in the same way as in Claim 3. Suppose the copies of P are denoted by $P^{(j)}$, $1 \le j \le m$, corresponding to GOTO exits $X^{(j)}_d$. The must-modify set of each copy is computed by

$$M^{P^{(j)}}_{\langle i_1,i_2,\ldots,i_{d-1}\rangle} = C\_OUT^{X^{(j)}_d}_{\langle i_1,i_2,\ldots,i_{d-1},l_d\rangle}. \quad (4)$$

In the above, we assumed conservatively that a GOTO exit may be taken as early as in the first iteration. This may not be true in some cases. However, since our analysis is insensitive to the exact content of an IF condition, the assumption is necessary.

After the above treatment, GOTO exits no longer affect our further discussion except in trivial details. For convenience, we will assume in the rest of the paper that GOTO exits do not exist.

Based on the equations developed so far, we present in Figure 4 the main algorithm for determining the existence of nonprivatizable definitions. A compiler may apply the algorithm to a DO loop at any loop level. The algorithm examines array privatizability for that loop as well as the inner loops (immediately nested or otherwise). We assume that all subroutine calls are in-lined so that no interprocedural propagation is necessary.

3.4 Traversal of the flow graphs

In this subsection, we discuss how to propagate cover sets through the flow graphs. Certain properties of cover sets allow us to simplify the propagation considerably. Most noticeably, we may break cycles in a condensed control flow graph before its traversal.

Definition If $\langle P_2, P_1\rangle$ is an edge in a control flow graph G such that P_1 dominates P_2, $\langle P_2, P_1\rangle$ is called a *back edge* in G [ASU86].

Claim 4 *If P_1 dominates P_2 in a condensed control flow graph G of the body of a DO loop at level d, then $C\_OUT^{P_1}_{\langle i_1,i_2,\ldots,i_d\rangle} \subseteq C\_IN^{P_2}_{\langle i_1,i_2,\ldots,i_d\rangle}$.*

Algorithm 1

Given: (1) DO loop L and all its inner loops (immediately nested or otherwise). Let $\mathcal{L}$ denote the set of these loops. (2) The index variables and the bounds of the outer loops of L, if any.

Output: For each loop in $\mathcal{L}$, an answer to the question whether there exist nonprivatizable definitions of A.

1. Call *Compute-Block-Sets*(L) (Figure 5) which recursively constructs the incoming-cover-sets, the outgoing-cover-sets, the past-modify sets and the must-modify sets in L and the inner loops.
2. **for** each DO loop L' in $\mathcal{L}$, **do**
   (a) **for** each (non-DO) statement S within L', **do**
   Compute the cover set $C^S_{\boldsymbol{\alpha}}|_d$ according to Claim 3, where $\boldsymbol{\alpha} = \langle i_1, i_2, \ldots, i_n\rangle$ is the index vector of the DO loops enclosing S and d is the loop level of L'.
   (b) Compute $UE_{\langle i_1,i_2,\ldots,i_d\rangle}$, the set of the upward exposed A elements in L_d, as prescribed in Claim 2.
   (c) Determine whether nonprivatizable A definitions exist in L_d as prescribed in Claim 1.

Figure 4: The main algorithm

Claim 5 *If $\langle P_2, P_1\rangle$ is a back edge in G, then $C\_IN^{P_1}_{\langle i_1,i_2,\ldots,i_d\rangle}$ remains the same if $\langle P_2, P_1\rangle$ is deleted from G.*

Our traversal algorithm (Figure 6) first prunes G according to Claim 5 and then propagates cover sets through the pruned graph. If G is *reducible* [ASU86], the pruned graph will be acyclic, in which each node needs to be visited only once. If G is irreducible, the traversal algorithm partitions the pruned graph into maximum strong components (MSC) and then visits the MSCs in a topsort order. The nodes in each MSC, Ω, are traversed iteratively. Each node will be visited $\#(\Omega)$ times. For a sparsely connected MSC, many of these visits could be avoided. However, we will not pursue the details.

4 The Live Array Analysis

As mentioned in Section 2, live array analysis is useful in dealing with the copy-out problem. The purpose of the analysis is to determine whether the self dependent definitions (SDDs) in a loop are live outside the loop, i.e. whether they reach any uses outside the loop.

Procedure Compute-Block-Sets(L)

Parameters: L, a DO loop nested in L_1, L_2, ..., L_{d-1} whose index variables are i_1, i_2, ..., l_{d-1}, respectively. The index variable of L is i_d.

1. For each DO loop, L', immediately nested in L, call *Compute-Block-Sets*(L').
2. Construct the condensed control flow graph, G, for the loop body of L.
3. Initialize $C\_IN^T_{\langle i_1,i_2,...,i_d\rangle} = C\_OUT^T_{\langle i_1,i_2,...,i_d\rangle}$ to empty sets, where T is the start node in G.
4. Traverse G according to Algorithm 2 (in the next subsection) and compute $C\_IN^P_{\langle i_1,i_2,...,i_d\rangle}$ and $C\_OUT^P_{\langle i_1,i_2,...,i_d\rangle}$ for every node P in G.
5. Compute the past-modify set $\tilde{M}^L_{\langle i_1,i_2,...,i_d\rangle}$ and the must-modify set $M^L_{\langle i_1,i_2,...,i_{d-1}\rangle}$ according to Claim 3.

Figure 5: Visit loops bottom-up

The merit of such information is twofolds. First, if no SDDs are live, then we are not concerned about the last defining definitions, because no copy-out is necessary. Second, if we can limit the copied-out elements to a minimum, run time overhead will be reduced. However, we want to avoid a full analysis of reaching definitions. Recall that the information we need is quite limited. We only need to know whether SDDs reach *any* outside uses and do not need to identify the reached uses. Moreover, we perform the live array analysis for the privatizable arrays only. We describe one scheme for the live array analysis in this paper. Two other schemes are described in [Li92].

The UEE scheme

Consider privatizable SDDs of array A in loop L_{d+1} at loop level $d+1$. L_{d+1} is represented by a condensed node in G_d, where G_d is the condensed flow graph at the d-th level as defined in Section 3. We use G_0 to denote the condensed control flow graph whose condensed nodes include all outermost DO loops in the program. G_0 is thus the flow graph at the highest level. Let $SM^{L_{d+1}}_{\langle i_1,i_2,...,i_d\rangle}$ denote $\bigcup_{l_d\le k\le u_d} D_{\langle i_1,i_2,...,i_d,k\rangle}$ which is the set of A elements defined by SDDs in L_{d+1}.

In the UEE scheme, we examine the set of the upward exposed elements of any condensed node L in G_d, which is denoted by $UEE^L_{\langle i_1,i_2,...,i_d\rangle}$. The set is determined by the following equation.

$$UEE^L_{\langle i_1,i_2,...,i_d\rangle} =$$

Algorithm 2

Given: (1) A DO loop, L, at level d. (2) The condensed flow graph, G, of the loop body. (3) The dominance relations in G.

Output: The incoming-cover-set and the outgoing-cover-set of each node in G.

1. Delete all back edges from G and obtain the *pruned* graph, G'.
2. For each node in G', compute its must-modify set.
3. Find the maximum strong components (MSCs) in G' and construct the *reduced* graph G'', which is acyclic. Each node in G'' represents an MSC in G'.
4. Traverse G'' in a topsort order. **for** each node visited, **do**
   (a) If the node corresponds to a single node P in G', then compute the incoming-cover-set and the outgoing-cover-set of P according to the basic flow equations Eq1 and Eq2.
   (b) If the node corresponds to an MSC, Ω, of several nodes in G', then call procedure *Iterate*(Ω).

Figure 6: The traversal algorithm

$$\bigcup_{l_{d+1}\le k\le u_{d+1}} (\tilde{M}^L_{\langle i_1,i_2,...,i_d,k\rangle} \cap UE_{\langle i_1,i_2,...,i_d,k\rangle}) \quad (5)$$

where $\tilde{M}^L_{\langle i_1,i_2,...,i_d,k\rangle}$ is the past-modify set of L as determined in Claim 3, and $UE_{\langle i_1,i_2,...,i_d,k\rangle}$ is the set of the upward exposed elements in the loop body of L, as determined in Claim 2. Note that these sets are already computed before the live array analysis. We also compute $UEE^B_{\langle i_1,i_2,...,i_d\rangle}$, the upward exposed elements in each regular node B in G_d, which is straightforward.

We now compute the UEE set for every node in G_k that is reachable from L_{k+1}, where $0 \le k \le d$. We then intersect the set $SM^{L_{d+1}}_{\langle i_1,i_2,...,i_d\rangle}$, defined above, with each of the UEE sets. If every intersection is empty, then we are sure that no copy-out is necessary. Otherwise, we conservatively assume that copy-out is necessary.

The motivation behind the UEE scheme is a hypothesis that if the SDDs in a DO loop do not reach any outside uses, then the array either disappears outside the loop or the array, as a temporary, is reinitialized in different places. In the latter case, any use of the array outside the loop will likely read a value that is "locally" defined. The scheme is conservative because

Procedure Iterate(Ω)

for each P in Ω, **do** /* initialize the cover sets */

- Suppose P_1, P_2, ..., P_m are the predecessors of P *outside* Ω, initialize $C\_IN^P_{\langle i_1,i_2,\ldots,i_d\rangle}$ to $\bigcap_{j=1,m} C\_OUT^{P_j}_{\langle i_1,i_2,\ldots,i_d\rangle}$.
- If P has no predecessors outside Ω, then initialize $C\_IN^P_{\langle i_1,i_2,\ldots,i_d\rangle}$ to $\top$, which is the set of *all* A elements.
- Initialize $C\_OUT^P_{\langle i_1,i_2,\ldots,i_d\rangle}$ to $C\_IN^P_{\langle i_1,i_2,\ldots,i_d\rangle} \cup M^P_{\langle i_1,i_2,\ldots,i_d\rangle}$.

for each P in Ω, **do** /* propagate the cover sets */

1. $SnowBall[P] := C\_OUT^P_{\langle i_1,i_2,\ldots,i_d\rangle}$.
2. $Givers := \{P\}$.
3. $Idle :=$ {All nodes in Ω but P}.
4. **while** $Idle$ is nonempty, **do**
   for each Q in $Givers$, **do**
   (a) $Receivers := \{ R \mid R \in Idle, \langle Q, R\rangle$ is an edge in $\Omega\}$.
   (b) $Idle := Idle - Receivers$.
   (c) **for** each node R in $Receivers$, **do**
   if $C\_IN^R_{\langle i_1,i_2,\ldots,i_d\rangle} \not\supseteq SnowBall[Q]$
   then $C\_IN^R_{\langle i_1,i_2,\ldots,i_d\rangle} := C\_IN^R_{\langle i_1,i_2,\ldots,i_d\rangle} \cap SnowBall[Q]$.
   $SnowBall[R] := C\_OUT^R_{\langle i_1,i_2,\ldots,i_d\rangle} := C\_IN^R_{\langle i_1,i_2,\ldots,i_d\rangle} \cup M^R_{\langle i_1,i_2,\ldots,i_d\rangle}$.
   $Givers := Receivers$.

Figure 7: Traversal of an MSC

even if the intersection of SM and an UEE (of node P) is nonempty, it is possible that the elements in the intersection are killed in every path from L_{d+1} to P.

The UEE scheme works well with the programs in Table 2. In TRACK and MDG, the privatizable arrays are local variables of subroutines. Within each subroutine, there are no references to the privatizable arrays beyond the loop nest. In OCEAN, a privatizable array CWORK is used in all three loops listed in Table 2. CWORK is also passed to a few subroutines called within those loops. However, CWORK is not upward exposed to the outside of any of those loops. The UEE scheme will hence report the definitions of CWORK as dead. Therefore, no copy-out is necessary. In TRFD, two privatizable arrays, XIJ and XRSIQ, are subroutine parameters. The uses of the two arrays outside the subroutines involve array subscripts that have complicated symbolic terms. The difficult subscripts, not the UEE scheme, force a conservative decision to copy out the two privatized arrays.

5 Summary

We have presented compiler algorithms for array privatization, a program transformation technique that is critical in the successful program parallelization in recent experiments. Where appropriate, we also report the behavior in real programs that we have examined. Although the analysis for array privatization is more complicated than commonly known data dependence analysis, the significant performance improvement it brings seems to well justify the additional compile time.

A review of the algorithms in Section 3 yields the following time complexity analysis. The time consumed by the algorithms can be divided into two parts. The first part is spent on constructing the condensed flow graphs and partitioning the graphs into maximum strong components. This part is proportional to the number of basic blocks in the program. The second part is spent on performing symbolic set operations to derive the cover sets and the modify sets. For a program whose flow graph is reducible, the number of such operations is proportional to the number of statements multiplied by the maximum number of loop levels, assuming the number of predecessors of each node in the flow graph is bounded by a small constant. If a program has components whose flow graphs are irreducible, an extremely rare case, then the second part is further multiplied by the maximum number of graph nodes in such components. The time for each symbolic set operation depends on the complexity of array references. Set representations are discussed in [Li92]. Due to lack of space, we do not explore further. In the loops listed in Table 2, array references are quite simple. Set operations are therefore quite simple. However, the compiler does need machineries that can manipulate symbolic expressions in the subscripts. Interprocedural analysis is also often required.

We summarize the practical aspects in array privatization as follows. (1) The UEE algorithm in Section 4 seems adequate for live array analysis in practice. (2) The storage allocation scheme in Section 2 is more than what we need for the programs examined so far, because in practice, we have found privatizable arrays at one loop level only. (3) The last defining iterations are easy to determine in our experience. (4) The flow graphs of the loops encountered are all reducible. Therefore, Procedure Iterate (cf. Figure 7) may be unnecessary. (5) The set computations described in Section 3 seem adequate. Unioning and intersection of the sets do occur over control paths. Further, array definitions and uses may interleave in the statements. Therefore, a simple summary of the references over a

whole loop is insufficient. (6) The privatizable arrays range from one-dimensional to three-dimensional, and the array subscripts may be constants or may contain loop indices and other symbolic terms. (7) We do not consider the exact contents of IF conditions here. But in at least two programs, we have found the contents important.

We are currently building an interprocedural analyzer to experiment on aggressive transformations, including array privatization. We adopt a hierarchical approach to dealing with complexity of data flows in large programs. (A similar approach is proposed by Rosen in [Ros77] for scalar analysis.) This approach takes advantage of structures in high level languages, which makes data flow analysis and dependence analysis efficient. Array privatization fits naturally in this approach, because a condensed node is just one example of using structures. The algorithms presented here will be tested in our experimental compiler. Many efficiency vs. precision issues raised in this paper will be explored through experiments.

6 Acknowledgement

The author is grateful to his former compiler group colleagues at CSRD: Rudy Eigenman, Jay Hoeflinger, and Dave Padua. The group performed hand analysis and transformation on several programs, which led to the cited paper [EHLP91] and eventually motivated the author to pursue the study in this paper. The practical cases cited here constitute only a small part in that tremendous effort. The author takes the sole responsibility for any potential errors in the description of those cases. The author also thanks the reviewers who suggested to include practical cases in this paper.

References

[ABC+88] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for determining useful parallelism. In *Proc. of the 1988 ACM Int'l Conf. on Supercomputing*, pages 207–215, July 1988.

[All70] F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, New York, 1988.

[BT82] T. Belytschko and C. S. Tsay. Whamse: A program for three-dimensional nonlinear nonlinear structural dynamics. Tech. Rept. No. NP-2250, Dept. of Civil Engin., Northwestern Univ, Evanston, IL, Feb. 1982.

[CF87] R. Cytron and J. Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *Proc. of the 1987 Int'l Conf. on Parallel Processing*, pages 19–27, August 1987.

[Coc70] J. Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, 5(7):20–24, 1970.

[EB91] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. In *Proc. of the Int'l Conf. on Parallel Processing*, August 1991.

[EHJ+91] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua. Restructuring fortran programs for cedar. In *Proc. of the 1991 Int'l Conf. on Parallel Processing*, August 1991.

[EHLP91] 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.

[Fea88] P Feautrier. Array expansion. In *Proc. of the 1988 ACM Int'l Conf. on Supercomputing*, pages 429–441, July 1988.

[GS90] 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.

[Li92] Z. Li. Array privatization: A loop transformation for parallel execution. Tech. Rept. No. 9226, Dept. of Computer Science, Univ. of Minnesota, April 1992.

[LT88] A. Lichnewsky and F. Thomasset. Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer. In *Proc. of the 1988 Int'l Conf. on Supercomputing*, pages 396–405, July 1988.

[PER89] M. berry et al. the PERFECT club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, 1989.

[Ros77] B. K. Rosen. High-level data flow analysis. *Communication of the ACM*, 20(10):712–724, 1977.

[TIF86] R. Triolet, F. Irigoin, and P Feautrier. Direct parallelization of CALL statements. In *Proc. of SIGPLAN '86 Symp. Compiler Construction*, pages 176–185, July 1986.