Automatic Array Privatization [⋆]

Peng Tu and David Padua

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

Abstract. Array privatization is one of the most effective transformations for the exploitation of parallelism. In this paper, we present a technique for automatic array privatization. Our algorithm uses data flow analysis of array references to identify privatizable arrays intraprocedurally as well as interprocedurally. It employs static and dynamic resolution to determine the last value of a lived private array. We compare the result of automatic array privatization with that of manual array privatization and identify directions for future improvement. To enhance the effectiveness of our algorithm, we develop a goal directly technique to analysis symbolic variables in the present of conditional statements, loops and index arrays.

1 Introduction

Enhancing parallelism, balancing load and reducing communication is among the major tasks of today's parallelizing compilers. Memory-related dependence can severely limit the potential parallelism of a program. Privatization is a technique that allows each concurrent thread to allocate a variable in its private storage such that each thread accesses a distinct instance of the variable. By providing a distinct instance of a variable to each processor, privatization can eliminate memory related dependence. Previous studies on the effectiveness of automatic program parallelization show that *privatization* is one of the most effective transformations for the exploitation of parallelism [8]. A related technique called *expansion* [13] transforms each reference to a particular scalar into a reference to a vector element in such a way that each thread accesses a different vector element. When applied to an array, expansion creates a new dimension for the array.

Because the access to a private variable is inherently local, privatization reduces the communication and facilitates data distribution. Since private instances of a variable are spread among all the active processors, privatization provides opportunities to spread computation among the processors and improve load balancing [16].

[⋆] The research described was supported by Army contract DABT63-92-C-0033 and CSRD affiliate program from Motorola Corporation. This work is not necessarily representative of the positions or policies of the Army or the Government.

Previous work on eliminating memory-related dependence focused on scalar expansion [19], scalar privatization [3], scalar renaming [6], and array expansion [13] [9]. Recently there have been several papers on array privatization [11][12][16].

We present an algorithm for automatically generating an annotated parallel program from a sequential program represented by a control flow graph. In the target parallel program, each loop is annotated with its privatizable arrays and their last value assignment conditions. The algorithm has been implemented in the POLARIS parallelizing compiler. Our work on automatic array privatization presents the following new results:

- We use data flow-based analysis for array reference. Compared with the dependence analysis-based approach [12], which has to employ parametric integer programming in its most general case, our approach is more efficient and can handle nonlinear subscripts that cannot be handled by integer programming.
- The algorithm proceeds from the bottom up, which allows us to easily extend the algorithm to program call trees for interprocedural analysis. Our experience shows this interprocedural array reference analysis is necessary in many cases for successful array privatization in real applications.
- We distinguish private arrays whose last value assignments can be determined statically from those whose last values have to be assigned dynamically at runtime. This work can potentially identify more private arrays than other algorithms can identify.
- To evaluate its effectiveness, we test the algorithm on the programs in the Perfect Benchmarks. We compare the automatic privatization with manual privatization described in a previous study[8]. We find that for further improvement, more sophisticated symbolic analysis techniques are needed.
- To facilitate further improvement, we develop a goal-directed technique to analyze symbolic variables in the present of conditional statements, loops, and index arrays.

The rest of the paper is organized as follows. Section 2 is an overview of the issues in automatic array privatization and gives an example that motivates this work. Section 3 presents the algorithm. The algorithm is divided into two parts: private array identification and last-value assignment resolution. Section 4 contains the experiments of automatic privatization of the Perfect Benchmarks and presents a comparison of automatic privatization with manual privatization. Section 5 presents a goal-directed technique that uses the SSA form of a program to determine symbolic values in the presence of conditional statements, loops, and index arrays. Section 6 presents the conclusion.

2 Background

Data dependence [2] specifies the precedence constraints in the execution of statements in a program due to data producer and consumer relationships. *Anti-*

dependence and *output dependence* are memory-related or *false* dependence because they are not caused by the flow of values from one statement to another, but by the reuse of memory locations. Consider the loop:

```
S1:  DO I = 1, N
S2:     A(1) = X(I,J)
S3:     DO J = 2, N
S4:         A(J) = A(J-1)+Y(J)
S5:     ENDDO
S6:     DO K = 1, N
S7:         B(I,K) = B(I,K) + A(K)
S8:     ENDDO
S9:  ENDDO
```

Because every iteration of loop *S1* accesses the same elements of array A, loop *S1* cannot be executed in parallel. However, there is no flow of value across iterations. The conflict can be resolved by declaring A to be private to each iteration of loop *S1*. We add the following directives to the loop:

```
C$DIR  INDEPENDENT
C$DIR  PRIVATE A(1:N)
C$DIR  LAST VALUE A(1:N) WHEN (I.EQ.N)
```

The *INDEPENDENT* directive is borrowed from HPF [10]. It specifies that the iterations of loop $S1$ are independent. There are two directives for a private array. The *PRIVATE* directive associates the privatizable arrays with each iteration of a loop. The *LAST VALUE* statement specifies the conditions when a processor should copy its private array value to the global array. The interpretation of the directives is as follows:

- Each processor cooperating in the execution of the loop allocates the private arrays in its local storage before executing any statement in the loop.
- During the entire execution of an iteration, references to a private array are directed to the processor's local instance.
- After the execution of an iteration has completed, the processor checks the last-value assignment condition. If the condition is satisfied, the processor copies the private array to the corresponding global array. This operation is called *copy-out*.

The results of our research on manual array privatization of Perfect Benchmarks didn't provide any case where a privatizable array needs both a local value and a global value. Hence our model does not have *copy-in*, that is, we do not allow values to be copied from the global array to the private array. Under this assumption, our definition of privatizable array is as follows.

Definition 1. Let A be an array that is referenced in a loop L. We say A is privatizable to L if the following conditions are satisfied.

1. Every fetch to an element of A in L must be preceded by a store to the element in the same iteration of L.
2. Different iterations of L may access the same location of A. □

The conditions for copying out the value of a private array to a global array are also determined by the compiler. In simple cases such as the one above, the algorithm can find a closed form for the condition. We call these cases *static last-value assignment.* In the more complicated cases, such as in the following loop, the last-value assignment has to be determined at run time:

```
C$DIR  INDEPENDENT
C$DIR  PRIVATE A(1:N)
C$DIR  LAST VALUE A(1) WHEN (I.EQ.N)
C$DIR  LAST VALUE A(2:N) WHEN DYNAMIC
S1:  DO I = 1, N
S2:     A(1) = X(I,J)
S3:     IF (A(1).GT.0) THEN
S4:        DO J = 2, N
S5:           A(J) = A(J-1)+Y(J)
S6:        ENDDO
S7:     ENDIF
S8:  ENDDO
```

In this example, the array section `A(2:N)` is conditionally assigned. `A` is still privatizable because it satisfies the privatizability conditions, but its last-value assignment cannot be determined at compile time. We use the key word *DYNAMIC* to specify that run-time resolution techniques such as *synchronization variable* [20] will have to be used for the array section `A(2:N)`. These cases are termed *dynamic last-value assignment.* For instance, the compiler can associate the subarray `A(2:N)` with a synchronization variable *last-iteration*, which stores the last iteration that was written to `A(2:N)`. Every iteration that defines `A(2:N)` will atomically compare its iteration number with the last iteration. If its iteration number is larger than the last iteration, the processor stores its iteration number into the last-iteration variable and copy-out `A(2:N)`. Otherwise, the assignment is ignored, because a later iteration has already written to `A(2:N)`. Note that because all the iterations are independent, the number of copy-out operations can be reduced by scheduling the loop backward from the last iteration to the first iteration.

3 Algorithm for Array Privatization

We consider the problem of identifying privatizable arrays in a data flow framework. From this point of view, to determine if an array is privatizable in a loop is to determine whether all its reaching definitions are coming from the same iteration of the loop.

3.1 Data Flow Framework

Problem Formulation. Data flow analysis examines the flow of values through a program and solves data flow problems by propagating information along the paths of a control flow graph. Because private arrays are associated with *DO* loops in the program, we must extend the traditional program flow graph with information about the scope of do loops.

Definition 2. $G = (N, E, s)$ be a program flow graph where N are nodes, E are arcs, and $s \in N$ is the initial node. Let L be a subflowgraph corresponding to a do loop (including all loops nested within it). We define as $control(L) \subset L$ the subset of nodes in L corresponding to the loop entry, increment and test of the loop index. $control(L)$ identifies the loop index, its limits and its step. We define the $body(L)$ as $L - control(L)$. □

When a program has nested loops, the control and body of the inner loop are included in the loop body of the outer loop. When the control flow of an inner loop is not important for the analysis of an outer loop, we can use abstraction for the inner loop and simplify the flowgraph by collapsing the subflowgraph corresponding to the inner loop into one node.

Definition 3. Let $G = (N, E, s)$ be a flow graph and L be a do loop in G. The $COLLAP(G, L)$ is a flow graph with the subflowgraph L collapsed into one node. □

Given a subflowgraph L corresponding to a loop, we want to determine if for every iteration of the loop, all reaching definitions to an array use come from the same iteration. We can do this through *def-use* analysis. The data values to be analyzed include both scalar values and array values. They are *scalar variable*, *subscripted variable*, and *subarray*. A subscripted variable consists of an array identifier and a list of subscript expressions. It is a special case of scalar variables. A subarray consists of a subscripted variable and one or more *ranges* for some of the indices in the subscript expression. A range includes expressions for the lower bound, upper bound, and stride. The notion of subarray we use in this paper is an extension at the *regular section* used by others[4]. Using subarray, we can represent the triangular region and banded region, as well as the strip, grid, column, row, and block of an array. For instance, the following examples respectively represent a dense upper triangle, grids in the upper triangle, and diagonal of array A.

```
(A(I,I:N),[I=1:N])
(A(I,I:N:2),[I=1:N:2])
(A(I,I),[I=1:N])
```

A range in a subarray is interpreted as a *FORALL*, `(A(I,I:N),[I=1:N])` is interpreted as `FORALL (I=1:N) A(I,I:N)`. Here the *FORALL* is just an assertion, no operational constraint, such as the order of assignment to different elements, is imposed.

We now describe the algorithm to do def-use analysis involving arrays. We start by computing *outward exposed* definitions and uses for each basic block S in the loop body. A definition of variable v in a basic block S is said to be outward exposed if it is the last definition of v in S. A use of v is outward exposed if S does not contain a definition of v before this use [22].

Definition 4. Let S be a basic block and VAR be the set of scalar variables, subscripted variables, and subarrays in the program. Henceforth these are called variables.

1. $DEF(S) := \{v \in VAR : v \text{ has an outward exposed definition in } S\}$
2. $USE(S) := \{v \in VAR : v \text{ has an outward exposed use in } S\}$
3. $KILL(S) := \{v \in VAR : v \text{ has a definition in } S\}$ □

For the flow information of a basic block S, we define $MRD_{\text{in}}(S)$ as the set of variables that are always defined upon entering S, $MRD_{\text{out}}(S)$ as the set of variables that are always defined upon exiting S. Let *pred*(S) be the set of immediate predecessors of S in the loop's flow graph ignoring all the back edges, $MRD_{\text{in}}(S)$ can be computed using the following equations:

$$MRD_{\text{in}}(S) = \cap_{t \in pred(S)} MRD_{\text{out}}(t) \tag{1}$$

$$MRD_{\text{out}}(S) = (MRD_{\text{in}}(S) - KILL(S)) \cup DEF(S) \tag{2}$$

We start from a conservative initial solution, with each MRD_{in} an empty set ϕ. The back edges in the graph are removed because $MRD(S)$ is only concerned with the values that are defined in the statements prior to S in the flow graph. Because back edges are deleted, the algorithm actually works on the DAG of the flow graph. Since back edges for inner loops do carry information for the analysis of the outer loop, they are handled by abstraction and aggregation in the next section.

The value of each set defined above such as DEF, USE, $KILL$, and MRD, is a subset of VAR. Hence the domain of the data flow information set is the powerset $\mathcal{P}(VAR)$. The effect of a $\cup$ (union) operation is to form a union of its operands. It is precise in the sense that it will not summarize two sets unless the summary set has exactly the same members as the two sets. For instance, $\{\texttt{A(I)} \cup \texttt{A(1 : N)}\}$ will return $\{\texttt{A(I)}, \texttt{A(1 : N)}\}$ unless $\texttt{I} \in [\texttt{1 : N}]$, but $\{\texttt{A(1 : N : 2)} \cup \texttt{A(2 : N - 1 : 2)}\}$ will return `A(1:N)`. The effect of a $\cap$ (join) operation is to form a join of its operands. It is conservative in the sense it will return an empty set ϕ if it cannot determine the join of its operands. For instance, $\{\texttt{A(I)} \cap \texttt{A(1 : N)}\}$ will return ϕ unless $\texttt{I} \in [\texttt{1 : N}]$. Because the join is conservative, there will be some potential loss of information at each join point of the flow graph. The effectiveness of the algorithm will hence depend on the system's ability to determine the relationship between symbolic variables. This issue will be discussed in Sec. 4.

An iterative algorithm for solving the MRD equation is shown in Fig. 1 as phases 1 and 2. Phases 3 and 4 are explained below.

```
Algorithm Privatize
    $privatize := func(L)$
    Input: flowgraph for loop $L$ with back edges removed
    Output: $DEF(L), USE(L), PRI(L)$

Phase 1: Collect local information
    foreach statement $S \in body(L)$ in rPostorder do
        if $S \in control(M)$ for some loop $M$ nested in $L$ then
            ! $S$ is in an inner loop, visit $M$ first
            $[DEF(S), USE(S)] \leftarrow privatize(M)$
            ! collapse all nodes in $M$ onto $S$
            $L \leftarrow COLLAP(L, M)$
        else
            compute local $DEF(S), USE(S)$
        endif
    endfor

Phase 2: Solve the MRD Data Flow Equations for each statement
    forall $S \in body(L)$ initialize $MRD(S) \leftarrow \phi$
    foreach $S \in body(L)$ in rPostorder do
        $MRD_{\mathrm{in}}(S) \leftarrow \cap_{t \in pred(S)} MRD_{\mathrm{out}}(t)$
        $MRD_{\mathrm{out}}(S) \leftarrow (MRD_{\mathrm{in}}(S) - KILL(S)) \cup DEF(S)$
    end

Phase 3: Compute Summary Sets for the Loop Body
    $DEF_{\mathrm{b}}(L) \leftarrow \cap_{t \in exits(body(L))}(MRD_{\mathrm{out}}(t))$
    $USE_{\mathrm{b}}(L) \leftarrow \cup_{t \in body(L)}(USE(t) - MRD_{\mathrm{in}}(t))$
    $PRI_{\mathrm{b}}(L) \leftarrow (\cup_{t \in body(L)} USE(t)) - USE_{\mathrm{b}}(L)$
    $PRI_{\mathrm{b}}^{\mathrm{st}}(L) \leftarrow DEF_{\mathrm{b}}(L) \cap PRI_{\mathrm{b}}(L)$
    $PRI_{\mathrm{b}}^{\mathrm{dy}}(L) \leftarrow PRI_{\mathrm{b}}(L) - PRI_{\mathrm{b}}^{\mathrm{st}}(L)$

Phase 4: Return aggregated set $DEF(L)$ and $USE(L)$
    test if it is profitable to privatize $PRI_{\mathrm{b}}(L)$
    determine last value assignment
    $[PRI^{\mathrm{st}}(L), PRI^{\mathrm{dy}}(L)] \leftarrow aggregate(PRI_{\mathrm{b}}^{\mathrm{st}}, PRI_{\mathrm{b}}^{\mathrm{dy}}, control(L))$
    $[DEF(L), USE(L)] \leftarrow aggregate(DEF_{\mathrm{b}}(L), USE_{\mathrm{b}}(L), control(L))$
    return $[DEF(L), USE(L)]$
```

Figure 1. Algorithm for Identifying Privatizable Arrays

Abstraction for Inner Loop. When the algorithm finds a loop nested inside a loop body, it will recursively call itself on the inner loop. To hide the control flow of an inner loop, we introduce some abstraction and extend the previous definition from a basic block to a complete loop. We start by defining the information for one iteration of the loop.

Definition 5. Let L be a loop and VAR be the variables in the program. We define the following set as *summary set* for *body*(L).

1. $DEF_{\mathrm{b}}(L) := \{v \in VAR : v$ has a MRD reaching all exits node of *body*(L) $\}$
2. $USE_{\mathrm{b}}(L) := \{v \in VAR : v$ has an outward exposed use in *body*(L) $\}$
3. $KILL_{\mathrm{b}}(L) := DEF_{\mathrm{b}}(L)$
4. $PRI_{\mathrm{b}}(L) := \{v \in VAR :$ every use of v has a reaching MRD in *body*(L) $\}$
   $\square$

The summary set is an abstraction of the effect of a loop iteration on the data flow values. Using the summary set, we can ignore the structure of the inner loops in the analysis of the outer loop. The trade-off is that we have to make a conservative approximation and may lose information in the process.

- $DEF_{\mathrm{b}}(L)$ is the *must define* variables for one iteration of L; i.e. the must define variables upon exiting the iteration:

$$DEF_{\mathrm{b}}(L) = \cap(MRD_{\mathrm{out}}(t) : t \in exits(L)) \tag{3}$$

- $USE_{\mathrm{b}}(L)$, the *possibly outward exposed use* variables, is the set of variables that are used in some statements of L, but do not have an MRD_{in} in the same iteration:

$$USE_{\mathrm{b}}(L) = \cup(USE(t) - MRD_{\mathrm{in}}(t)) : t \in body(L) \tag{4}$$

- The *privatizable variables* are the variables that are used and not exposed to definitions outside the iteration:

$$PRI_{\mathrm{b}}(L) = \cup\{USE(t) : t \in body(L)\} - USE_{\mathrm{b}}(L) \tag{5}$$

In the analysis of the outer loop, we must consider the total effect of an inner loop on data flow values. That is, we need to account for the effect of back edges and index domain of the loop. We can do this by listing the summary set for each iteration of the loop. We will use an approximation called *aggregated set* to compute $DEF(L)$, $USE(L)$, $KILL(L)$, and $PRI(L)$. The aggregation computes the region spanned by each array reference in $USE_{\mathrm{b}}(L)$, $DEF_{\mathrm{b}}(L)$, $KILL_{\mathrm{b}}$, and $PRI_{\mathrm{b}}(L)$ across the iteration space. Because we only consider do loops, the aggregation is a relatively straightforward interpretation of loop index and boundaries in the do-entry of the loop. In our representation of variables, a subarray is represented as a subscripted variable together with a subscript range. To aggregate a subarray, we just need to concatenate the loop index and boundaries with the subscripted variable of subarray. For instance, if `I` is a loop index or an induction variable with value `[1:N:1]`, then `A(I,J)` will be aggregated as `(A(I,J),[I=1:N]) = A(1:N,J)` and `A(I,1:I)` will be aggregated as `(A(I,1:I),[I=1:N])`.

Because one iteration's use may only be exposed to the definitions in some previous iterations of the same loop, a naive aggregation of $USE_{\mathrm{b}}(L)$ may exaggerate the exposed use set. The reason is that the uses covered by the definitions in previous iterations are not exposed to the outside of the loop, and therefore they should be excluded from the aggregated $USE(L)$ set. For instance, in

```
   DO I = 2, N
S1:   A(I) = A(I-1) + B(J)
   ENDDO
```

the information for one iteration is $USE_b(L) = \{$`A(I-1)`, `B(J)`, `J`$\}$ and $DEF_b(L) = \{$`A(I)`$\}$, the region aggregately defined in all iterations prior to the ith iteration is `A(2:I-1)`, and `A(I-1)` is exposed to definitions outside the loop only in the first iteration, that is, $USE(L) = \{$`A(1)`$\}$.

Profitability of Privatization. After an array is identified as privatizable in a loop, we need to determine if different iterations of a loop will access the same location of the array. For instance, in the following loop:

```
S1:  DO I = 1, N
S2:     A(I) = ...
S3:     ...  = A(I)
S4:  ENDDO
```

the algorithm will identify that `A(I)` is privatizable. We can privatize `A(I)` using a private scalar as follows:

```
C$DIR INDEPENDENT
C$DIR PRIVATE X
C$DIR LAST VALUE A(I) = X
S1:  DO I = 1, N
S2:     X   = ...
S3:     ... = X
Sn:  ENDDO
```

This transformation is useful for conventional compiler optimization. Today's optimizing compilers usually will not allocate a register to a subscripted variable `A(I)` in the original program because they have very limited capability to disambiguitize the array reference. In the transformed program, it is easy for them to allocate a register to a scalar `X`. The transformation can also reduce the amount of *false sharing* in multiprocessor caches. In a distributed memory system with *owner computes* rule [21][5] [14], the transformed program effectively transfers the ownership of `A(I)` to iteration I; hence the processor scheduled to execute the iteration I can execute operations in `S2` even if it does not own `A(I)`. This transformation can facilitate data distribution to reduce communication and improve load balance [16].

For the purpose of eliminating memory-related dependence in this paper, the array `A` in the previous example need not be privatized. The condition for privatization exists when different iterations of the loop access same location. This can be determined by examining $PRI_b(L)$. We will call the test the *profitability test*. Let *A(r)* be a reference to array A where r is a subscript expression if $A(r)$ is a subscripted variable, or a range list if $A(r)$ is a subarray.

If $A(r)$ is a subscripted variable and r is a monotonic function of loop index i, then different iterations of i will access different locations of $A(i)$; hence it is not profitable to privatize $A(r)$, otherwise it is profitable. When there is more than one subscript of A in $PRI_\mathrm{b}(L)$, we need to test if there is dependence between each pair of subscripted variables. We can use the Banerjee Test [2] to determine if within the loop boundaries two references referred to the same location. If $A(r)$ is a subarray, we need to determine if there is an iteration $j \neq i$ such that $A(r) \cap A(r[i/j]) \neq \phi$, where $r[i/j]$ represents r after we substitute each appearance of loop index i with j. Again one has to test for each pair of occurrences if there is more than one occurrence of subarrays. This discussion is summarized in the algorithm shown in Fig. 2.

```
Algorithm Profitability Test
    Input: $PRI_\mathrm{b}$ for loop $L$: with index $i \in [p : q : t]$
    Output: $PRO$, arrays profitable for privatization

    $PRO \leftarrow \phi$
    foreach $A \in PRI_\mathrm{b}$ do
        $ALL_A \leftarrow \{A(r) : A(r) \in PRI\}$
        foreach pair $A(x), A(y) \in ALL_A$ — where $x$ and $y$ can be the same
            let $X \leftarrow$ set of values in $x$
            let $Y \leftarrow$ set of values in $y$
            if $(\exists j \in [p : q : t] | j \neq i, X[i/j] \cap Y \neq \phi)$
                $PRO \leftarrow PRO + A$
            !Notice that if $x = y$ and $x$ does not contain $i$, the test is satisfied.
        endfor
    endfor
```

Figure 2. Profitability Test

3.2 Last Value Assignment

Live Analysis. *Live analysis* is needed to determine if a privatizable variable is live after exiting the loop. If it is live, the last-value assignment will be necessary to preserve the semantics of the original program; otherwise no last-value assignment is needed for that variable. A last-value assignment statement can be ignored when the private array is not used after the loop, or there are subsequent definitions of the array before any use.

Definition 6. Let S be a node in the flowgraph. The live variables at the bottom of S are the set of variables that may be used after control passes the bottom of S. We define

1. $LVBOT(S) := \{v \in VAR : v$ may be used after $S\}$
2. $LVTOP(S) := \{v \in VAR : v$ may be used after S or in $S\}$ □

Let $succ(S)$ be the set of immediate successors of S in the program flowgraph. The equations for $LVTOP$, $LVBOT$ are

$$LVBOT(S) = \cup_{t \in succ(S)} LVTOP(t) \tag{6}$$

$$LVTOP(t) = (LVBOT(S) - KILL(S)) \cup USE(S) \tag{7}$$

The algorithm traverses the flow graph backward and uses the aggregated set for each loop. This algorithm is just the natural extension of scalar live analysis to include array references.

Static and Dynamic Last-Value Assignment. After live analysis, we can ignore the last-value assignments for private arrays that are not live at the bottom of the loop. However, the remaining live private arrays have to be copied to their global counterparts. Two problems prevent static determination of iteration that copies its private array to the global array. One, as shown in our early example, is due to conditional definition. Without information about which branch the program will take at runtime, it is impossible to determine which iteration shall assign the last value. Another problem is that some complicated subscript expressions make it inefficient to compute at compile time which iteration will assign the last value. In these cases, we will use well-known run-time techniques such as [20] to resolve the output dependence.

Our first step is to identify the private arrays that need dynamic last-value assignments because of conditional definition. PRI_{b} contains all the array uses that are covered by some definition in the same iteration of the loop; some of the uses are conditional, where they are covered by some conditional definition. DEF_{b} contains all the variables that must be defined in every iteration of the loop. Therefore, $PRI_{\mathrm{b}}^{\mathrm{st}} = PRI_{\mathrm{b}} \cap DEF_{\mathrm{b}}$ contains the privatizable arrays that are unconditionally defined. Hence $PRI_{\mathrm{b}}^{\mathrm{dy}} = PRI_{\mathrm{b}} - PRI_{\mathrm{b}}^{\mathrm{st}}$ contains the conditionally defined privatizable arrays.

Because of the profitability test, at least one element of the array in $PRI_{\mathrm{b}}^{\mathrm{st}}$ is defined in several iterations. To determine for each iteration what element has to be copied back to the global array, we define a *write back set* as the sections of private array that have to be copied back to the global array for iteration i.

Definition 7. Let L be a loop body and PRI^{st} be the static private arrays. The Write Back Set (WBS) of L for iteration i is defined as the sections of arrays in PRI^{st} that are written in the ith iteration, but are not written thereafter. $\square$

From the definition we can compute the WBS by comparing the set defined in iteration i and the set defined in the iterations after i. The algorithm is shown in Fig. 3.

Note that the last iteration of loop L will always write back all its static private arrays. When we cannot find a closed form for WBS, we can move the array to $PRI_{\mathrm{b}}^{\mathrm{dy}}$ and use run-time resolution. Actually the algorithm itself can be linked into the program to perform a run test for each iteration. In most cases, the algorithm will find a closed form and therefore WBS can be determined at

Algorithm Write Back Set

```
    Input: $PRI_b^{st}$ for loop $L$: with index $i \in [p:q:t]$
    Output: $WBS$, for iteration $i$

    $WBS \leftarrow \phi$
    foreach array $A \in PRI_b^{st}$ do
        $ALL_A \leftarrow \{A(r) : A(r) \in PRI_b^{st}\}$
        $WBS \leftarrow ALL_A - \cup_{j \in [i+t:q:t]} ALL_A[i/j]$
    endfor
```

Figure 3. Compute Write Back Set

compile time. The following example shows how the algorithm in Fig. 3 works in two different situations.

```
S1:  DO I = 1, N
S2:     DO J = 1, M
S3:        A(J) = ...
S4:        B(I+J) = ...
S5:     ENDDO
        ...
Sn:  ENDDO
```

For loop `S1`, $PRI_b^{st} = \{\mathtt{A}(1:\mathtt{M}), \mathtt{B}(\mathtt{I}+1:\mathtt{I}+\mathtt{M})\}$. `A(1:M)` will be accessed in all iterations after a given `I<N` because `A(1:M)` does not depend on `I`. Hence WBS for `A` in iteration $\mathtt{I} \neq \mathtt{N}$ is ϕ, the empty set. Only the last iteration of loop `S1` will copy out `A(1:M)`. For `B`, `B(I+1:I+M)` is in ALL_B for iteration `I`, `B((I+1)+1:M+N)` is modified in iterations from `I+1` to `N`, hence the WSB for `B` is `B(I+1)`.

3.3 Interprocedural Analysis of Privatizable Arrays

In many cases, we need to do interprocedural analysis for array privatization. We can find more deeply nested loops by looking at the loops in the subroutines. To use the algorithm for interprocedural analysis, we generalize the loop flow graph to incorporate subroutine bodies.

Definition 8. Let R be a subroutine and VAR be the variables in the subroutine. We define the *subroutine summary set* for R as follows:

1. $DEF(R) := \{v \in VAR : v$ has a MRD reaching all exits node of $R\ \}$
2. $USE(R) := \{v \in VAR : v$ has an outward exposed use in $R\ \}$
3. $KILL(R) := DEF(R)$ □

The algorithm to find the *subroutine summary set* is the same used above to compute DEF_b, USE_b, and $KILL_b$. The input to the algorithm is now the flowgraph of the subroutine. We run the algorithm in bottom-up order on the program call tree, such that each time we encounter a subroutine call in a program, the summary set for the subroutine has already been computed. When the algorithm finds a subroutine call node, it reads the summary set of the subroutine, simplifies the summary set to get rid of variables that are not visible to the caller, and maps the formal parameters and common variables in the summary set to the corresponding actual parameters and common variables at the caller. Because array reshaping usually occurs in programs written in FORTRAN, the array defined in the subroutine may have a different shape. Our interprocedural mapping program will linearize an array in the subroutine if it has a different number of dimensions as in the caller. After the specialization, the algorithm will use the subroutine summary set for the call statement.

We implemented the algorithm with interprocedural analysis in the POLARIS system. It allows us to do automatic array privatization in loops with subroutine calls.

4 Automatic versus Manual Array Privatization

To evaluate the effectiveness of the algorithm, we ran the automatic array privatization on the Perfect Benchmarks. We compared the number of private arrays found by the algorithm with that of the manual array privatization reported in [8]. The result is shown in Table 1. The first column reports the number of private arrays identified by both manual and automatic privatization. The second column reports the number of private arrays identified by manual privatization but not by automatic privatization. The third column reports the number identified by automatic privatization but not by manual privatization. By comparing

Table 1. Number of Private Arrays

Program	Automatic and Manual	Manual Only	Automatic Only
ADM (AP)	2	12	0
ARC2D (SR)	0	2	0
BNDA (NA)	12	3	4
DYFESM (SD)	0	1	11
FLO52 (TF)	0	0	4
MDG (LW)	17	1	1
MG3D (SM)	1	4	0
OCEAN (OC)	4	3	0
QCD (LG)	22	7	0
SPEC77 (WS)	25	14	0
TRACK (MT)	20	2	0
TRFD (TI)	4	0	0

the results of automatic privatization and manual privatization, we found that the algorithm is sufficient to discover most of the privatizable arrays. The lattice for array references is also adequate for representing the array use and definition in the programs of Perfect Benchmarks. Where our algorithm failed, we found that in most instance it is due to lack of information about symbolic variables. Some of the ambiguities can be resolved by a more powerful forward substitution algorithm than that available in our current system. For instance, our algorithm failed to identify private array `XE` in subroutine *solvhe* of *DYFESM*. In this case, `XE` is defined in the *geteu* subroutine as a two-dimensional array `XE(NDDF,NNPED)` and used in *solvhe* as a one dimensional array `XE(NDFE)`. It turns out that `NDFE = NDDF*NNPED` after interprocedural forward substitution.

Some of the ambiguities can be resolved by enhancing the traditional scalar constant propagation and forward substitution. For instance, a common difficulty is conditionally defined loop boundaries. Such a situation occurs in subroutine *initia* of *ARC2D* code. Two common variables `JLOW` and `JUP` are defined as follows:

```
  IF (.NOT.PERIDC) THEN
     JLOW = 2
     JUP = JMAX - 1
  ELSE
     JLOW = 1
     JUP = JMAX
  ENDIF
```

These two common variables are then used in subroutine *filerx, filery* as loop boundaries:

```
L1: DO N = 1, 4
L2:    DO J = JLOW, JUP
          DO K = KLOW, KUP
             WORK(J,K,1) = ...
          ENDDO
       ENDDO
       IF (.NOT.PERDIC) THEN
L3:       DO K = KLOW, KUP
             WORK(1,K,1) = WORK(2,K,1) + ...
             WORK(JMAX,K,1) = WORK(JMAX-1,K,1)
          ENDDO
       ENDIF
       ...
    ENDDO
```

For the array `WORK` to be privatized in loop `L1`, we need to determine the reference `WORK(2,K,1)` and `WORK(JMAX-1,K,1)` in loop `L3` are defined in `L2`. If we inspect the condition in the *IF* statement, we can determine that when `L3` is executed the condition is `(.NOT.PERDIC)`. Under the same condition, `JLOW=2, JUP=JMAX-1`;

hence the use is covered by the definition in `L2`. A simpler method is to propagate the upper bound of `JLOW=2` and lower bound of `JUP=JMAX-1`, and we do not need to interpret the condition of *IF* statement.

A problem of similar nature happens in *BDNA*. It involves the bound of an induction variable L:

```
L1: DO I = 2, NSP
L2:    DO J = 1, I-1
          XDT(J) = ...
       ENDDO
       L = 0
L3:    DO J = 1, I-1
          IF (IND(J).NE.0) THEN
             L = L+1
          ENDIF
       ENDDO
L4:    DO J = 1, L
          ... = XDT(J)
       ENDDO
    ENDDO
```

For `XDT` to be private to loop `L1`, the use in `L4` must be covered by definition in `L2`, i.e., we need to know if `L<=I-1`. Computing the upper bound for `L` in loop `L3`, the validity of this condition in `L4` can be confirmed.

The idea of keeping more than one possible value for scalars can be generalized to propagate the value for each element of an array. This is very useful in the case of subscripted subscripts. For instance, in *ARC2D*, array `JPLUS` and `JMINU` are used to store the neighboring element on a ring of size `JMAX`. That is, `JPLUS(I) = I+1 mod JMAX`, and `JMINU(I) = I-1 mod JMAX`. These values are used throughout the program. By propagating the value of the whole array, we can use our algorithm to compute the range defined and used.

Another interesting case arises in `MDG`, where we must inspect the condition of an *IF* statement to determine whether the array `RL` can be privatized in subroutine *poteng* and *interf*:

```
    DO I = 1, NMOL1
L2:    DO J = I+1, NMOL
          KC = 0
          DO K = 1, 9
             RS(K) = ...
C1:          IF (RS(K).GT.CUT2) THEN
                KC = KC + 1
             ENDIF
          ENDDO
          DO K = 2, 5
C2:          IF (RS(K).LE.CUT2) THEN
                 RL(K+4) = ...
                ENDIF
             ENDDO
C3:          IF (KC.EQ.0) THEN
                DO K = 11, 14
                   ... = RL(K-5)
                ENDDO
             ENDIF
          ENDDO
       ENDDO
```

Note that whenever (KC.EQ.0) in C3 is true, (RS(K).LE.CUT2) must also be true, because if (RS(K).GT.CUT2), then KC must be greater than 0 due to the increment in C1. Hence RL(6:9) is privatizable in L2 because whenever RL(6:9) is used, it refers to the values defined in the same iteration.

In some rare cases, user direction is needed to determine if an array is privatizable. This happens in *OCEAN*, where it cannot be statically determined if the array C and CA are defined in subroutine *in*. The *in* writes to the C and CA, but the definitions are surrounded by an error condition test. Without knowing whether the error condition will abort the program, the algorithm has no way of knowing those arrays are defined whenever the program returns from subroutine *in*.

5 Demand Driven Symbolic Analysis

In the last section, we showed that to determine the region of an array that is used in a program, a major task is to determine the relationship between symbolic variables. In this section, we present a demand-driven technique to determine the value relationship between symbolic variables. This technique is based on the Static Single Assignment (SSA) form. SSA is an intermediate representation of a program that has two useful properties:

1. Each use of a variable is reached by exactly one definition to that variable.
2. The program contains PHI functions that merge the values of a variable from a distinct incoming control-flow graph.

[1, 15, 17] present various applications of SSA, and [7] deals with efficiently transforming programs into SSA form.

5.1 Determine Symbolic Value on Demand

The SSA form can be used to track the value of symbolic variables on demand. For instance, from the following SSA representation of a piece of code in *DYFESM*, we can determine that the value of NDFE_1 used in L3 is NDDF_1*NNPED_1 since NDFE_1 is assigned in S1.

```
S1: NDFE_1 = NDDF_1 * NNPED_1
    ...
    DO K_1 = 1, N_1
L1:    DO I_1=1, NDDF_1
L2:       DO J_1=1, NNPED_1
             XE(I_1,J) = ...
          END DO
       END DO
L3:    DO I_2 = 1,NDFE_1
          ... = XE(I_2)
       END DO
    END DO
```

The loop boundaries for `L1,L2` are the same `NDDF_1` and `NNPED_1`. Hence the use of `XE` in `L3`, which is `XE(1:NDFE_1)=XE(1:NDDF_1*NNPED_1)`, is covered by the definition `XE(1:NNDF_1,1:NNPED_1)` in `L1`. Because in a SSA representation, each use of a variable can be reached by exactly one assignment to that variable, it is very easy to track the value of a variable by its name.

In the traditional forward substitution, because it is difficult to know which variable should be forward substituted, it usually substitutes all the variables in a program and rolls back later to avoid redundant computation. In contrast, the backward tracking of a value through SSA variable name is done on demand.

5.2 Dealing with Conditionals

One difficulty in dealing with a conditional statement is to propagate the condition from the assignment of a variable to the use of the variable. For the example code from *ARC2D*, it is difficult to know that the condition guarding the assignment to `JLOW_1` is the same condition guarding the use of `JLOW_1`. In situation like this, the unique variable name in the SSA representation serves as a handle to link the scattered information.

The SSA representation for the *ARC2D* example is as follows (we only show in SSA form for variables involved in loop boundaries):

```
    IF (.NOT.PERIDC) THEN
       JLOW_1 = 2
       JUP_1 = JMAX - 1
    ELSE
       JLOW_2 = 1
       JUP_2 = JMAX
    ENDIF
    JLOW_3 = PHI(Cond(.NOT.PERIDC),JLOW_1,JLOW_2)
    JUP_3 = PHI(Cond(.NOT.PERIDC),JUP_1,JUP_2)

L1: DO N = 1, 4
L2:    DO J = JLOW_3, JUP_3
```

```
            DO K = KLOW, KUP
               WORK(J,K,1) = ...
            ENDDO
         ENDDO
S1:      IF (.NOT.PERDIC) THEN
L3:         DO K = KLOW, KUP
               WORK(1,K,1) = WORK(2,K,1) + ...
               WORK(JMAX,K,1) = WORK(JMAX-1,K,1)
            ENDDO
         ENDIF
         ...
      ENDDO
```

Note that the PHI functions for JLOW_3 and JUP_3; it is inserted in the program to distinguish values of a variable from different branches of the control flow graph, in this case the different branches of a conditional statement. We use an extended PHI function to include the predicate for the conditional statement. Cond(.NOT.PERIDC) specifies the condition in the IF statement, if Cond(.NOT.PERIDC) is true, JLOW_3 will take the value of the second parameter of the PHI function which is JLOW_1, if Cond(.NOT.PERIDC) is false, it will take the value of the third parameter, JLOW_2.

We will first show how to interpret the conditional statement and then show how to compute the upper and lower bounds for the variables. For loop L3 to be executed, the condition (.NOT.PERDIC) must be true since L3 is control dependent on the S1. Tracing the values of JLOW_3,JUP_3 to the PHI function we know that they will have value JLOW_1,JUP_1. Tracing the value of JLOW_1,JUP_1 further, we have JLOW_3=JLOW_1=2,JUP_3=JUP_1=JMAX-1. Since now the value of JLOW_3=2,JUP_3=JMAX-1 matches the value for the subscript of the WORK(2,K,1) and WORK(JMAX-1,K,1), we do not need to trace back the value for JMAX any further. At this point we can compare the array region WORK(JLOW_3:JUP_3, KLOW:KUP, 1) defined in loop L2, which is WORK(2:JMAX-1, KLOW:KUP, 1), with the array regions WORK(2, KLOW:KUP, 1) and WORK(JMAX-1, KLOW:KUP, 1). The definition covers the uses, and WORK array is privatizable to loop L1.

As discussed before, in this example it is sufficient to prove WORK is privatizable just by showing the lower bound of JUP_3 is greater than or equal to JMAX-1 and the upper bound of JLOW_3 is less than or equal to 2. To compute the bounds for a variable, we can make a conservative choice at each PHI function and ignore the predicate for the conditional statement. We start by tracing back the values for variables until they are unified. Then we take the max or min on the second and third parameter of a PHI function and ignore the predicate.

```
max(JLOW_3) = max(PHI(Cond(.NOT.PERIDC),JLOW_1,JLOW_2))
   = max(PHI(Cond(.NOT.PERIDC),2,1)) = max(2,1) = 2

min(JUP_3) = min(PHI(Cond(.NOT.PERIDC),JUP_1,JUP_2))
   = min(PHI(Cond(.NOT.PERIDC),JMAX-1,JMAX)) = JMAX-1
```

In addition to being goal directed and on demand, the backward tracing scheme can stop the tracing when the symbolic expressions in question are *unified*, i.e., when the variables in the expressions are the same. After that, the additional unwinding of values will not gain any more information. In a forward propagation scheme, everything must start from the most primitive variables and in the case of several levels of conditionals, the number of branches may quickly explode and complexity may grow out of control. Because the backward tracing is goal directed and incremental, we can easily set complexity constraints such as the maximum backward tracing level to a fix number of nested PHI functions. After that, the algorithm can give up and degrade gracefully to reduce compile time.

5.3 Bounds for Monotonic Variables

The value of an induction variable or a monotonic variable depends on the structure of the loop in which it is assigned. Induction variable's last value can be determined using induction variable substitution technique such as presented in [18]. In this section, we will show a technique to estimate the bounds of monotonic variable.

Using SSA form of loop L3 in the example from *BDNA*, we have:

```
       L_1 = 0
L3:    DO J = 1, I-1
          L_2 = PHI(L3,L_4,L_1)
          IF (IND(J).NE.0) THEN
             L_3 = L_2 + 1
          ENDIF
          L_4=PHI(Cond(IND(J).NE.0),L_3,L_2)
       ENDDO
```

The original loop L4 will use L_2 as the value of L. In this example, we extend the PHI function to include loop label L3 in it to identify the loop control. Following the terminology used by Wolfe on induction variables[18], L_2 will appear as a *Strongly Connected Region (SCR)* that includes a loop header PHI function and some conditional PHI functions. To find the upper bounds of L_2, we need to find the cycle with maximum increment to L_2 in the SCR. Similarly, for the lower bound of a monotonic variable, we need to find the cycle with minimum increment to L_2 in the SCR. This can be accomplished by backward tracing and compute bounds on PHI function as follows:

```
L_2 = PHI(L3,L_1,L_4) = PHI(L3,PHI(Cond(IND(J).NE.0),L_3,L_2),0)
   = PHI(L3,PHI(Cond(IND(J).NE.0),L_2+1,L_2),0)
```

Hence the maximum value for L_2 will be:

```
   max(L_2)=max(PHI(L3,PHI(Cond(IND(J).NE.0),L_2+1,L_2),0))
      = 0+(I-1)*max_inc(PHI(Cond(IND(J).NE.0),L_2+1,L_2))
      = (I-1)*max(0,1)=I-1
```

The way to handle loop PHI function is to take its trip count and multiply the trip count to the maximum increment in the SCR. To find the maximum increment for a loop, we will choose the branch with maximum increment in a conditional PHI function. In the example, the monotonic variable is L_2 and its maximum increment for each iteration is 1. Lower bound for monotonic variables can be computed by taking a minimum function over the PHI functions.

5.4 Index Arrays

The use of the index array in the program makes it difficult to determine the array reference region in a program. ARC2D uses an index array JPLUS. It is assigned as follows:

```
L1:  DO J=1, JMAX
        JPLUS(J) = J+1
     END DO
     JPLUS(JMAX)=JMAX
     ...
L2:  DO J = 1, JMAX
        ... = ... FLUX(JPLUS(J),K,N)
     END DO
```

We can use the SSA representation to find out the value of JPLUS(J) in loop L2. We will extend the SSA representation to array in the following way: (1) create a new array name for each array assignment; (2) use the subscript to identify which element is assigned; (3) replace the assignment with a special PHI function that will be written as MU((subscript),assignment,old). The assignment A(I) = exp is converted to A_1 = MU((I), exp, A_0), which is interpreted as element A_1(I) will take the value of exp in the assignment while other elements of A_1 will take the value in the A_0 as before the assignment. Using this extension, our example can be transformed into the following SSA form:

```
L1:  DO J=1, JMAX
        JPLUS_2 = PHI(L1,JPLUS_1,JPLUS_0)
        JPLUS_1 = MU((J),J+1,JPLUS_2)
     END DO
     JPLUS_3 = MU((JMAX),JMAX,JPLUS_2)
     ...
L2:  DO J = 1, JMAX
        ... = ... FLUX(JPLUS_3(J),K,N)
     END DO
```

For subscript expression JPLUS_3(J) in loop L2, it can be evaluated as follows.

```
   JPLUS_3(J) = (MU((JMAX),JMAX,JPLUS_2))(J)
      = (MU((JMAX),JMAX,PHI(L1,JPLUS_1,JPLUS_0)))(J)
      = (MU((JMAX),JMAX,PHI(L1,MU((J),J+1,JPLUS_2),JPLUS_0)))(J)
      = (MU((JMAX),JMAX,MU(([1:JMAX]),J+1,JPLUS_0)))(J)
```

The expression can be interpreted as

```
JPLUS_3(J) = IF J=JMAX THEN
                JMAX
             ELSEIF J in [1:JMAX] THEN
                J+1
             ELSE
                JPLUS_0(J)
             ENDIF
```

Note that the PHI function for loop L1 defines an aggregated region of the index array.

6 Conclusion

We presented an algorithm to automatically identify privatizable arrays in sequential FORTRAN programs to eliminate memory-related dependence. The algorithm has been implemented in the POLARIS system to perform interprocedural array privatization. Our experiments have thus far indicated that the algorithms can privatize most of the arrays privatized by hand in [8]. To increase the coverage of the algorithms, it seems necessary to use more sophisticated techniques for determining the equivalence of symbolic variables, and interprocedural symbolic values and bounds propagation. To this purpose, we proposed a goal-directed technique that uses the SSA form of a program to determine the values and bounds of symbolic variables in the presence of conditional statements, loops, and index arrays. We are currently implementing the symbolic analysis technique and studying the application of array privatization to data distribution for greater local access and better load balancing.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.
2. Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.
3. M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, pages 71–88, 1989.
4. D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

5. D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.
6. Ron Cytron and Jeanne Ferante. What's in a Name? or The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proc. 1987 International Conf. on Parallel Processing*, pages 19–27, August 1987.
7. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
8. R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In *Proc. 4-th Workshop on Programming Languages and Compilers for Parallel Computing.* Pitman/MIT Press, August 1991.
9. P. Feautrier. Array expansion. In *Proc. 1988 ACM Int'l Conf. on Supercomputing*, July 1988.
10. High Performance Fortran Forum. High performance fortran language specification (draft). Technical report, High Performance Fortran Forum, January 1993.
11. Zhiyuan Li. Array privatization for parallel execution of loops. In *Proc. of ICS'92*, pages 313–322, 1992.
12. D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Data dependence and data-flow analysis of arrays. In *Proc. 5rd Workshop on Programming Languages and Compilers for Parallel Computing*, August 1992.
13. D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.
14. A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. the SIGPLAN '89 Conference on Program Language Design and Implementation*, June 1989.
15. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computation. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.
16. Peng Tu and David Padua. Array privatization for shared and distributed memory machines. In *Proc. 2nd Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines, to appear on ACM SIGPLAN Notices 1993*, September 1992.
17. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
18. Michael Wolfe. Beyond induction variables. *ACM PLDI'92*, 1992.
19. Michael Joseph Wolfe. Optimizing supercompilers for supercomputers. Technical Report UIUCDCS-R-82-1105, Department of Computer Science, University of Illinois, October 1982.
20. Chuan-Qi Zhu and Pen-Chung Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering*, 13(6):726–739, June 1987.
21. H. Zima, H.-J. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.
22. Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1991.

This article was processed using the $\LaTeX$ macro package with LLNCS style