J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*

PROGRAM CONTROL STRUCTURES

A program control structure (or a control structure) in a programming language is a control statement and a collection of statements it controls. The execution order of statements in a program is determined by the combination of the sequential control and the control structures. A sequential control specifies the execution of statements in the order they appear in the source code (i.e., if statement S_1 appears immediately before statement S_2, the execution of S_1 should appear immediately before that of S_2). The control structures specify departures from the sequential control and include structures for the conditional control, the repetition control, and the invocation control. A conditional control selects a group of statements to execute when its corresponding control condition is true; it is usually represented by an IF statement. A repetition control specifies the execution over a group of statements many times; it is usually represented by a DO statement, a WHILE statement, a FOR statement, and so on. An invocation control, usually represented by a CALL statement, stops the current normal execution, starts executing the called procedure, and resumes the normal execution after the procedure is finished. These are common program control constructs. Other program control constructs, such as those specifying parallel execution of multiple computations of a program, are also used in some parallel languages.

To speed up the execution of computer programs, modern computers rely heavily on a technique known as parallel processing. Parallel processing executes several independent statements (or sections) of a single program at the same time. The more independent statements a program has, the faster this program can be run using parallel processing. One approach to finding independent statements of a program is to use a restructuring compiler to recognize them automatically. This process is also referred to as parallelization. Restructuring compilers for parallelization consider repetition control structures and check to see if the computations specified by these repetition structures, which take most of the execution time of a program, can be run in parallel.

Although control structures help a programmer to write a computer program in a structured and concise way, some structures, especially conditional control structures, can make it difficult for computer programs to run faster. Conditional control structures affect the exactness of program analysis, and therefore may reduce the effectiveness of a restructuring compiler. When a conditional control structure is involved in the recurrence relation of a loop, parallelizing such a loop is especially difficult. In the following sections, two approaches will be presented to parallelizing repetition control structures containing conditional control structures and invocation structures.

Repetition Control Structures

In general, the repetition control structures can be categorized into iterative DO loops (also called iterative loops or DO loops) and WHILE loops. A DO loop is a loop with a known index set, that is, the number of iterations that the loop has at run time is known to be fixed before the loop is run, and this number is independent of the calculation inside the loop. The DO loops defined here should not be confused with the loops constructed by DO statements as in the FORTRAN language, which may contain exit conditions causing the loops to be

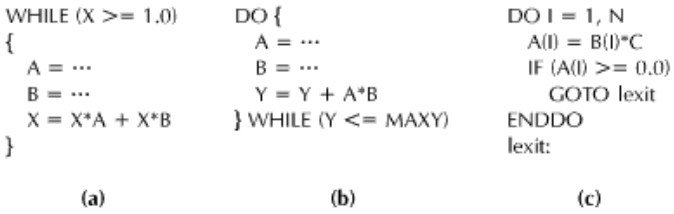

Fig. 1. Typical forms of WHILE loops.

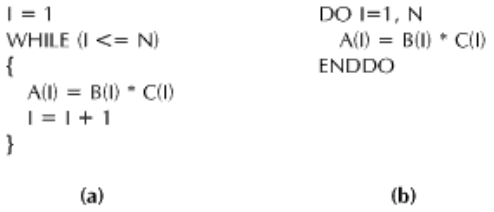

Fig. 2. Transforming a WHILE loop into a DO loop.

terminated earlier than specified by their index sets. The rest of the loops are WHILE loops whose number of iterations are dependent on the calculations of the loops.

While Loops. WHILE loops can be in many forms. Three typical forms are shown in Fig. 1. Figure 1(a) has a Boolean expression that specifies the loop control at the top of the loop, as in a C WHILE loop. This form of loop allows zero-trip execution. When the Boolean expression evaluates to false, the loop will finish its execution.

Figure 1(b) has a Boolean expression for loop control at the bottom of the loop, represented in a C DO-WHILE loop structure. This form of loop will execute its body at least once. At the end of each iteration, the Boolean expression is evaluated. If it evaluates to false, the loop will finish its execution.

The last form of loop, shown in Fig. 1(c), is represented by a FORTRAN DO loop with an exit IF statement that conditionally branches out of the loop. This form also allows zero trip (when $N < 1$).

Parallelizing these forms of loops in general is difficult because the number of iterations is dependent on the calculation of the loop. Recurrence relations occur often in these loops, and thus are also obstacles for parallelizing them. Some special WHILE loops, however, can be transformed into DO loops, and therefore can be handled by well-studied parallelization techniques for DO loops (1,2,3). For instance, the WHILE loop in Fig. 2(a) can be transformed into the DO loop shown in Fig. 2(b).

DO Loops. If a DO loop contains only assignment statements, as in Fig. 2(b), it is referred to as a *simple* DO loop. A DO loop may also contain conditional branching (IF statements). DO loops with conditional branching can be classified by the availability of the predicate values defining the branching. If conditional branching is based on variables that are not a part of the recurrence relation in the loop, the loop is referred to as a *with-IF* DO loop (or a *with-IF* loop). The following is such an example:

```
DO I = 1, N IF (A(I) > 0) THEN B(I) = C(I) + D(I) ELSE B(I) = C(I) − D(I) ENDIF ENDDO
```

A *conditional cyclic loop* (or conditional cyclic DO loop) is a DO loop with conditional branching in which the values of the variables upon which conditional branching is made are not known before executing the loop (i.e., some of the variables defining the predicates are evaluated within the loop itself). For instance, the following loop is a conditional cyclic loop:

```
DO I = 2, N IF (X(I−1) > 0) THEN X(I) = A(I) ELSE X(I) = B(I) ENDIF ENDDO
```

Note that these two forms of loops with conditional branching have the number of iterations that do not depend upon the calculations of the loops, and therefore they are not WHILE loops.

These forms of loops can be further categorized in terms of the presence of recurrence relations in loops. For example, a with-IF loop that contains recurrence relations is termed a recurrence with-IF loop. In the following sections, techniques for parallelizing the DO loops are discussed.

Parallelization of do Loops

For shared-memory multiprocessor systems, the iterations of a DO loop can be executed on different processors, and this can shorten the execution time of the loop. To make this parallel execution possible, data dependence in the original loop must be preserved in the parallel execution.

To preserve data dependence among different iterations running on different processors, one approach is to add communication and synchronization statements into the iterations. The loops that are run in parallel this way are generally referred to as DOACROSS loops. It is true that any loop can be converted to a DOACROSS loop, but this will not necessarily shorten the execution time.

For loops which do not have data dependence among different iterations, their parallel execution does not involve interaction among different iterations; therefore, these parallel loops are referred to as DOALL loops. Because of the absence of communication and synchronization statements, DOALL loops do not have overhead caused by such statements and are more likely to speed up the execution of the loops. Many DO loops can be converted to DOALL loops because they either do not have dependence among different iterations or can be transformed into DOALL loops using techniques such as the ones discussed in the following sections. In this article, we will discuss DOALL loops only.

Dependence Relations. *Data dependence* exists when a statement accesses the same memory locations as another statement. Consider two statements S_i and S_j when S_i precedes S_j on the control flow path of a given program. Several types of dependence are possible (1).

- S_j is data *flow dependent* on S_i if S_i writes to a memory location before S_j reads from that location.
- S_j is data *antidependent* on S_i if S_i reads from a memory location before S_j writes to that location.
- S_j is *output dependent* on S_i if S_i writes to a memory location before S_j rewrites to that location.

Control dependence arises if the execution of S_j is determined as a result of the control path chosen by the execution of a conditional statement S_i. For convenience, that S_j is dependent on S_i is denoted as $S_j \delta S_i$ without distinguishing the type of the dependence.

Along with dependence information, information on dependence distance (or direction) across the loop iterations is often helpful (4,5). Suppose that statements S_1 and S_2 are enclosed in a loop that has index i, and suppose that $S_{1_1}^i$ is an instance of S_1 when $i = i_1$. If $S_{2_2}^i$ is dependent on $S_{1_1}^i$, the dependence is denoted as $S_{2_2}^i \delta S_{1_1}^i$ and it is said to have a *dependence distance* $i_2 - i_1$ (4). If the dependence distance is positive, the dependence has a *direction vector* $(<)$ and is denoted as $S_2 \delta (<) S_1$. If the dependence distance is negative or

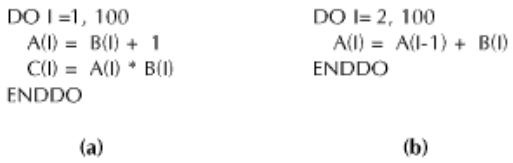

Fig. 3. Loop parallelization.

zero, the dependence has a direction vector (>) or (=), respectively. So, a dependence with its direction vector (<) or (>) is a cross-iteration dependence, which prohibits the iterations from being executed independently.

A *dependence graph* for a program is composed of directed edges connecting dependent operations as nodes. For parallel machines, the dependence graph plays the important role of exposing parallelism because edges in the graph impose a partial ordering of operations to be executed.

Parallelizing Loops without Recurrence. Consider an example in Fig. 3. We would like to run all the iterations of each loop in Fig. 3 in parallel, but in Fig. 3(b) the memory location being read in each iteration (except for the first iteration) was written in the previous iteration. Trying to run them in parallel could not preserve the data-dependence relations existing between the different iterations.

In Fig. 3(a), there are no data-dependence relations among the different iterations. Therefore, this loop is parallelizable. This loop, however, does contain a flow-dependence relation within each iteration. Because transforming loops into DOALL loops does not break these flow-dependence relations, this flow dependence does not affect the parallelization of the loop.

Any dependence relation within an iteration is called a loop-independent dependence relation, whereas any dependence relation among different iterations is called a loop-carried dependence relation (1). We can say that all the iterations of a loop can be run in parallel if and only if there is no loop-carried dependence in that loop.

In general, a conventional algorithm of loop parallelization checks all array-reference pairs in a loop. If there is no loop-carried dependence, this loop is parallelizable. This general scheme can be extended to loop nests. A loop in a loop nest is parallelizable if there is no loop-carried dependence corresponding to this loop. If all the loops in the loop nest are parallelizable, the loop nest is parallelizable.

For each array-reference pair, dependence equations are formed to detect whether a loop-carried dependence exists. The existence of integer solutions to the dependence equations shows the existence of dependence for this pair. Computing integer solutions is also referred to as solving the dependence equations in the integer domain. Most techniques assume that array subscripts and loop bounds are affine functions of loop indices. This assumption is reasonable and covers most of the loops in practice. Under this assumption, the dependence equations are equivalent to an integer programming problem. Unfortunately, integer programming is NP-complete. The general algorithm for integer programming, which is of exponential time complexity, is too expensive to be applied. As a result, many approximation algorithms, and even some exact algorithms under more strict constraints, are used in practice. The earliest algorithms, such as the greatest common divisor (*GCD*) test and the Banerjee test (6,7), are the most widely used. However, they only give the sufficient conditions for the nonexistence of dependence. If these conditions are true, then these algorithms prove the nonexistence of dependence. If these conditions are false, then they assume the existence of dependence without knowing whether a dependence really exists.

The GCD test gives the necessary and sufficient conditions for integral solutions to the dependence equations that exclude the constraints on loop bounds and direction vectors. The Banerjee test considers both

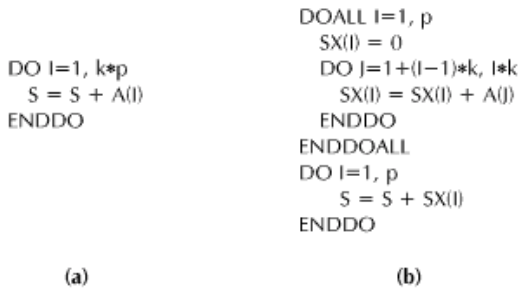

Fig. 4. Parallelization of reduction.

bounds and direction vectors, but in general, it generates the sufficient conditions for the nonexistence of dependence in the real domain. No real solutions in the Banerjee test means no integer solutions.

The Fourier-Motzkin algorithm (8) solves the dependence equations in the real domain as the Banerjee test does. It also enhances the Banerjee test by achieving the exact solutions in the real domain. It, however, does not exactly solve the problem in the integer domain. The Fourier-Motzkin algorithm in general can be expensive, but several reports in the literature (9,10) assert that it is reasonably efficient in practice.

Other approximation algorithms can also be found in Ref. 9. Some algorithms (see Ref. 9) can be used to get the exact solutions for special cases, whereas several other algorithms (10,11) solve an integer programming problem exactly. For dependence analysis, these algorithms for solving an integer programming problem have been shown to be reasonably efficient in practice.

The techniques discussed previously can handle loops that are part of simple loops or with-IF loops. When a loop contains recurrence relations, a technique particular for parallelizing recurrence may be used.

Parallelizing Loops with Recurrence. Recurrence occurs frequently in practice, and parallelizing it is very important. Recurrence appears if the computation of one iteration relies on the values computed in the previous iterations. A simple form of recurrence is reduction, which has been extensively studied (12). For example, a reduction loop in Fig. 4(a) can be converted to a parallel one in Fig. 4(b), where p is the number of processors. For other forms of recurrence, as in Fig. 3(b), some fast methods were created (12). These methods change the recurrence solving algorithm into one that exhibits more parallelism. Of course, these methods may need additional hardware support and may produce redundant computations.

Appearance of IF statements in loops may make parallelization of these loops even more difficult. If a with-IF loop has recurrences and its conditional branching is based on variables that are not a part of the recurrence relation in the loop, the loop is termed a recurrence-with-IF loop. If the recurrence is linear of order 1, it is called an R $< n,1 >$ with-IF loop, which can be parallelized (5).

A conditional cyclic loop is a with-IF loop with recurrence and its conditional branching is based on variables that are a part of the recurrence relation. It is difficult to parallelize a conditional cyclic loop because it is difficult to precompute possible values of the predicate. Conditional cyclic loops are not rare in sequential programs and present a major obstacle to automatic restructuring of nonnumerical programs for parallel processing (13).

In the following sections, we assume that programs are written in FORTRAN. We will discuss how a conditional cyclic loop is related to a Boolean recurrence. We then discuss parallelizing conditional cyclic loops based on a binary tree representation of the loops. For convenience, $\log n$ will denote $\log_2 n$ and will be assumed to have an integer value. The values of x/y and $\sqrt{x}$ will also be assumed to be integers. We also discuss an array data-flow analysis to recognize parallel loops that are not conditional cyclic loops. This approach can handle

call statements, IF statements, and symbolic variables; therefore, it is powerful enough to handle loops in the real programs.

Parallelizing Conditional Cyclic Loops

Conditional cyclic loops can be classified by how easily one can figure out all possible values of the variables that define the predicate of the IF statement in the loop. A mixed recurrence loop is a conditional cyclic loop where the statement S_i causes a recurrence by itself and the recurrence variables are the ones defining the predicate of the IF statement. So, the possible values of the variables are not known until the recurrences are solved. If the recurrences are linear, the loops are called linear mixed recurrence loops. In practice, nonlinear mixed recurrence loops are extremely rare; they never occurred in our experiment (13). A postfix-IF loop of order m is a conditional cyclic loop in which the data dependence $S_j\delta(<)S_i$ has a distance m, where S_i is an assignment statement that does not cause a recurrence by itself and S_j is an IF statement. In postfix-IF loops, the two possible values of the predicate variables (one for the true branch and the other for the false branch) at every iteration of the loop are available without the need to solve a recurrence relation [i.e., the possible values of the variables are "immediately" available (14)]. We consider parallelizing linear mixed recurrence loops only, of which postfix-IF loops are a special case.

Figure 5 shows some typical examples and (abstract) dependence graphs of cyclic loops with conditional branching, where $(\cdot)$ denotes a direction vector of singly nested loops. Figure 5(a) shows an example of an R $< n,1 >$ loop for which fast efficient parallel algorithms are known, whereas Figs. 5(b) and 5(c) show examples of conditional cyclic loops. If one can convert a conditional cyclic loop into a form of an R $< n,1 >$ loop by precomputing the possible values of the predicate of the branching, then conditional cyclic loop can be parallelized as well.

Consider the loop shown in Fig. 5(b), which is a postfix-IF loop of order 2. To determine the predicate value of the branching at a particular iteration, one needs to know the value of the array elements in the predicate. Because there are two possible choices for the array element value at each iteration, there can be four possible cases of the predicate evaluations at a particular iteration. These predicate evaluations can be expressed by the following Boolean equations:

$$
\begin{aligned}
b(1) &= [\mathbf{c}(2)\ .\mathrm{GT.}\ \mathbf{c}(1)] \\
b(2) &= [\mathrm{w}(3)\ .\mathrm{GT.}\ \mathbf{c}(2)] \cdot b(1)\ .\mathrm{OR.}\ [\mathrm{v}(3)\ .\mathrm{GT.}\ \mathbf{c}(2)] \cdot \bar{b}(1) \\
b(\mathrm{i}) &= [\mathrm{w}(\mathrm{i}+1)\ .\mathrm{GT.}\ \mathrm{w}(\mathrm{i})] \cdot b(\mathrm{i}-1) \cdot b(\mathrm{i}-2) \\
&\quad .\mathrm{OR.}\ [\mathrm{w}(\mathrm{i}+1)\ .\mathrm{GT.}\ \mathrm{v}(\mathrm{i})] \cdot b(\mathrm{i}-1) \cdot \bar{b}(\mathrm{i}-2) \\
&\quad .\mathrm{OR.}\ [\mathrm{v}(\mathrm{i}+1)\ .\mathrm{GT.}\ \mathrm{w}(\mathrm{i})] \cdot \bar{b}(\mathrm{i}-1) \cdot b(\mathrm{i}-2) \\
&\quad .\mathrm{OR.}\ [\mathrm{v}(\mathrm{i}+1)\ .\mathrm{GT.}\ \mathrm{v}(\mathrm{i})] \cdot \bar{b}(\mathrm{i}-1) \cdot \bar{b}(\mathrm{i}-2),\ (3 \le \mathrm{i} \le \mathrm{n}-1)
\end{aligned}
$$

where $\bar{b}$ denotes the negation of the Boolean expression b and "." represents Boolean AND. By solving the Boolean equations, one can parallelize the loop. As we can see from the example, the difficulty of parallelizing a conditional cyclic loop depends on how complex the Boolean equations are.

Towle (15) defined a B-postfix-IF loop, which has a very restricted form of Boolean equation, while Banerjee and Gajski (14) proposed a Boolean equation solving hardware for a general form of postfix-IF loops. No other attempts have been made to parallelize linear mixed recurrence loops, of which postfix-IF loops are a simple special case. In the following section, we discuss how complex the Boolean equations are in linear mixed recurrence loops.

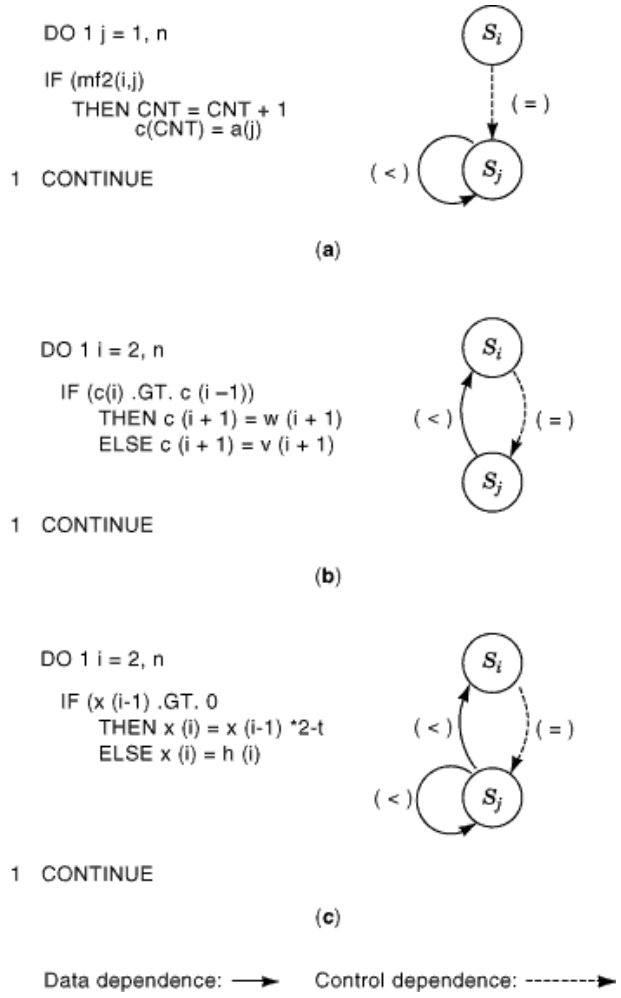

Fig. 5. Cyclic loops with conditional branching: (a) recurrence loop with IF; (b) post fix-IF loop of order 2; (c) linear mixed recurrence loop of order 1.

Boolean Recurrences and Conditional Cyclic Loops. Suppose that we have a conditional cyclic loop like the following:

L: DO1 $i = 2,n$ IF**c** $(x_{i-m}, \ldots, x_{i-1})$ THEN $x_i = \phi_i$ ELSE $x_i = \pi_i$ 1 CONTINUE

where ϕ_i and π_i are arbitrary functions. Although the expression **c** may contain any x_i $(1 \leq i \leq n)$, we use the notation $\mathbf{c}(x_{i-m}, \ldots, x_{i-1})$ to highlight dependence cycles between the predicate and the assignment statements. If ϕ_i and π_i are known values before executing the loop (i.e., constants or functions not involving x_i's), then L is a postfix-IF loop of size n. If ϕ_i and π_i are linear recurrences of x_i, then L is a linear mixed recurrence loop of size n.

Consider a set of Boolean variables $\{b_1, \ldots, b_i\}$ with an integer i. Let the 2^i *minterms* (a minterm is a Boolean AND of b_i's) of these variables be numbered 1, 2, ..., 2^i as they appear in a usual truth table, and

$P_t(b_1, \ldots, b_i)$ be the tth minterm. Then loop L can be represented by the following set of equations:

$$x_i = \begin{cases} \phi_i & \text{if } b_{i-1} = 1 \\ \pi_i & \text{if } b_{i-1} = 0 \end{cases} (2 \le i \le n)$$

(0 and 1 are Boolean constants)

where Boolean variable b_i is defined as the following nonlinear Boolean recurrence of order m:

$$b_i = \sum_{t=1}^{2^m} e_{i,t} P_t(b_{i-m}, \ldots, b_{i-1}), \quad (2 \le i \le n)$$

and $e_{i,t}$ represents a value of the Boolean expression $\mathbf{c}(x_{i-m}, \ldots, x_{i-1})$ for one of 2^m possible cases of the predicate evaluation based on x_k's $[(i - m) \le k \le (i - 1)]$. The value of the Boolean variable b_i depends on the paths chosen by the branching at the previous iterations, which are represented by the values of b_k's $[(i - m) \le k \le (i - 1)]$. So, every conditional cyclic loop has an imbedded Boolean recurrence.

To solve the Boolean recurrence, the coefficients $e_{i,t}$'s need to be evaluated first. However, to evaluate $e_{i,t}$'s for a linear mixed recurrence loop, we need to solve linear recurrences whose coefficients can be determined only after solving the Boolean recurrence. The straightforward way of breaking this circularity is to evaluate all the possible values of the linear recurrence variables. This leads to solving a full-order Boolean recurrence:

$$b_i = \sum_{t=1}^{2^{(i-1)}} e_{i,t} P_t(b_1, \ldots, b_{i-1}), (2 \le i \le n)$$

Evaluating all the possible values of the linear recurrence variables requires considering all possible branching decisions made at previous iterations. Thus, there are 2^{i-1} possible cases of evaluating the predicate $\mathbf{c}$ at a particular iteration i. The evaluation of $e_{i,t}$'s is considerably more complex than for postfix-IF loops.

Suppose that in loop L, $\phi_i = a_i * x_{i-1} + c_i$ and $\pi_i = \bar{a}_i * x_{i-1} + \bar{c}_i$, where a_i, c_i, $\bar{a}_i$, and $\bar{c}_i$ are constant coefficients. Then we have a linear mixed recurrence loop of order 1. Based on the idea of solving a full-order Boolean recurrence, the program can be changed as in Fig. 6. Loop L1 is for the precomputation of all possible values of x_i. If we consider loop L1 as an example for evaluating 2^{n-1} linear recurrences of size $n - 1$, this can be executed in $O(\log n)$ time with $p = (n - 1) \cdot 2^{n-1}$ processors using the idea in Ref. 16. Loop L2 can be done in a constant time with $p = 2^{n-1} - 1$ processors, assuming that the time for evaluating the expression $\mathbf{c}$ is constant. Because loop L4 is a first-order linear recurrence with an IF, it can be executed in $O(\log n)$ time with $p = (n - 1)$ processors (5). So, we could solve any linear mixed recurrence loop of order 1 in $O(\log n)$ time if we could solve the full-order Boolean recurrence, loop L3, in $O(\log n)$ time. However, by the "fan-in lemma" (17,18), which states that one cannot evaluate an expression of binary operations on n data in less than $\log n$ time even with an infinite number of processors, we cannot solve this full-order Boolean recurrence in $O(\log n)$ time because there are $n - 1$ expressions of $\Theta(2^n)$ variables to be evaluated, assuming that a processor can consume at most two operands at a time.

Binary Trees and Conditional Cyclic Loops. Because it is not desirable to parallelize a general conditional cyclic loop as long as one tries to solve directly the Boolean recurrence caused by the loop, another approach, which is based on a binary tree representation, can be considered as in the Boolean recurrence solver suggested for postfix-IF loops (14).

Consider loop L in the previous section. By having each node of a binary tree represent each possible value of branching predicate $\mathbf{c}$, and the two edges from each node represent the two branches of the IF statement,

```
/* precomputation of recurrence variables */
L1:          x'_{1,1} = x_1
             DO 1 i = 2, n
               DO 1 t = 1, 2^{i-1}
                 IF (t is odd)
                   THEN x'_{i,t} = ā_i * x'_{i-1,⌈t/2⌉} + c̄_i
                   ELSE x'_{i,t} = a_i * x'_{i-1,⌈t/2⌉} + c_i
1            CONTINUE

/* precomputation of Boolean coefficients */
L2:          DO 2 i = 1, n - 1
               DO 2 t = 1, 2^{i-1}
                 e_{i,t} = c(x'_{i-1,⌈t/2⌉}, ..., x'_{i-m+1,⌈t/2^{m-1}⌉})
2            CONTINUE

/* Boolean recurrence */

             b_1 = e_{1,1}
L3:          DO 3 i = 2, n - 1
                 b_i = Σ_{t=1}^{2^{i-1}} e_{i,t} · P_t(b_1, ..., b_{i-1})
3            CONTINUE

/* first-order linear recurrence with IF */
L4:          DO 4 i = 2, n
                 IF b_{i-1} THEN x_i = a_i * x_{i-1} + c_i
                            ELSE x_i = ā_i * x_{i-1} + c̄_i
4            CONTINUE
```

Fig. 6. Restructured equivalent of a linear mixed recurrence loop.

say the left edge for the false branch and the right edge for the true branch, loop L can be naturally represented by a binary decision tree of height $n - 1$.

Consider the complete binary tree of height $n - 1$. Let $e_{i,t}$ be the tth node from the left on the ith level of the tree (see Fig. 7). Then $e_{i,t}$ $(1 \leq t \leq 2^{i-1})$ represents a predicate value based on one of 2^{i-1} possible cases of evaluating **c** at the ith iteration of loop L. So, the execution of a conditional cyclic loop L is equivalent to forming a particular path from the root by selecting a node at each level of the tree, provided that the tree is already formed.

Selecting a path on the tree is basically a parallel prefix problem (19). Let Path_t $(1 \leq t \leq 2^{i-1})$ be the Boolean product of all the $e_{i,t}$'s on the path from the root to the tth leaf node. Then,

$$\text{Path}_1 = \prod_{i=1}^{n-2} \bar{e}_{i,1}$$

$$\text{Path}_2 = e_{n-2,1} \prod_{i=1}^{n-3} \bar{e}_{i,1}$$

$$\vdots$$

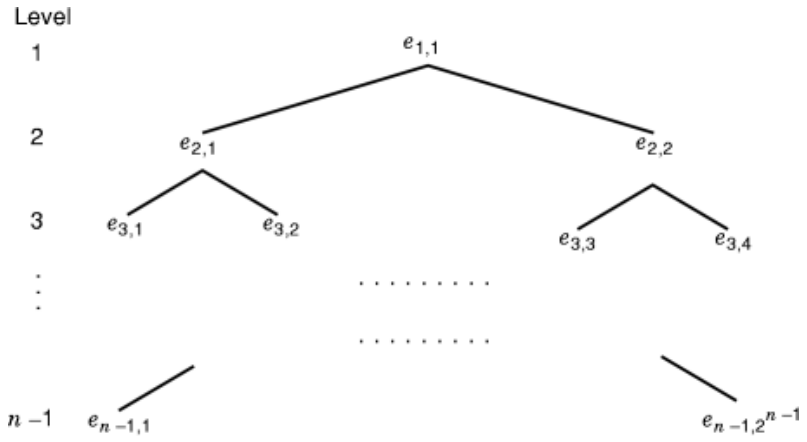

Fig. 7. Tree representation of conditional cyclic loop.

$$\text{Path}_{2^{n-1}-1} = \bar{e}_{n-2,2^{n-2}} \prod_{i=1}^{n-3} e_{i,2^i}$$

$$\text{Path}_{2^{n-1}} = \prod_{i=1}^{n-2} e_{i,2^i}$$

where $\bar{e}$ denotes the negation of the Boolean value e.

Suppose that a processor is assigned to each "mutually exclusive" complete subtree of height 2 of the binary tree from top to bottom (i.e., processors are assigned to the nodes on every other level of the tree starting from the root). By checking the value of the root of the subtree, each processor can determine which one of its two descendant nodes will be taken for the actual execution path. This produces $(2^n - 1)/3$ edges for the tree of height n. With these edges, a tree of reduced height can be formed.

Consider a binary tree of height 4 as shown in Fig. 8. Suppose one processor is assigned to the root node and four processors to the nodes at level 3. Thus, there are five subtrees of height 2 to be checked in parallel. In each subtree, by checking the value of a parent node, one can determine which one of the two descendant nodes should be included in the execution path if the parent node is a part of the execution path. Assuming five edges (E_1 to E_5) are the resulting edges as in Fig. 8, a reduced tree is formed as follows. Because edge E_1 is taken at the root node, edges E_4 and E_5 cannot be a part of the execution path. So, they are excluded. $E_2(E_3)$ becomes a left(right) son of E_1, because in the original tree the parent node in $E_2(E_3)$ is a left(right) son of the descendant node in E_1. The resulting tree's height becomes half of the original (see Fig. 8). This tree reduction is essentially a step of Boolean product in parallel.

Algorithm PATH (Path finding for a conditional cyclic loop) /* the value of every $e_{i,t}$ is known */ /* $P_{i,t}$ is an ordered set of nodes */ /* $P_{1,1}$ is the output */ L1: DO 1 $k = 1$, $\log(n-1)$ L2: DOALL 2 $j = 1$, $(n-1)/2^k$ $i = (j-1)2^k + 1$ L3: DOALL 3 $t = 1$, 2^{i-1} IF ($k = 1$) THEN IF $e_{i,t}$ THEN $P_{i,t} = \{e_{i,t}\} \cup \{e_{i+1,2t}\}$ ELSE $P_{i,t} = \{e_{i,t}\} \cup \{e_{i+1,2t-1}\}$ ELSE BEGIN $e_{i+2^{k-1}-1,x} =$ the last entered element of $P_{i,t}$ IF $e_{i+2^{k-i}-1,x}$ THEN $P_{i,t} = P_{i,t} \cup P_{i+2^{k-1},2x}$ ELSE $P_{i,t} = P_{i,t} \cup P_{i+2^{k-1},2x-1}$ END 3 CONTINUE 2 CONTINUE 1 CONTINUE

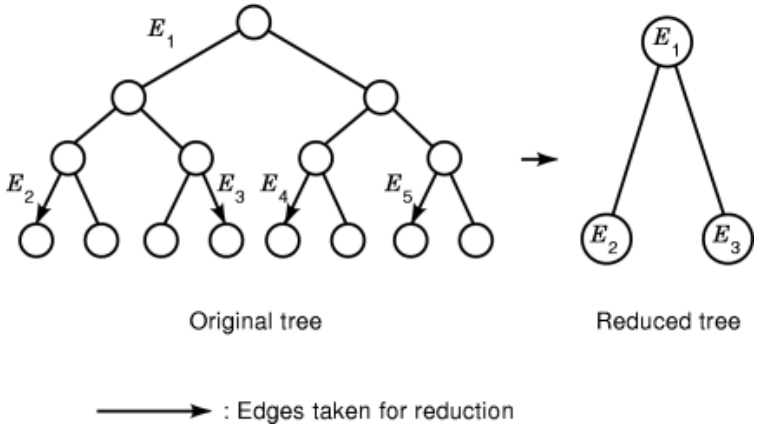

Fig. 8. Tree reduction.

Repeating the preceding tree reduction recursively until the tree is reduced to a single node gives the sequential execution path in $O(\log n)$ time. The algorithm for finding a "correct" execution path in the binary tree representation of a conditional cyclic loop is described as Algorithm PATH. Notice that when the number of available processors $p = 1$, Algorithm PATH is equivalent to the sequential execution of a conditional cyclic loop. The correctness of the algorithm can be easily checked by induction.

To use Algorithm PATH to parallelize a linear mixed recurrence loop, we need to precompute the possible values of $e_{i,t}$'s to set up the tree representation of the loop. For a linear mixed recurrence loop of order m and of size n, the precomputation is equivalent to solving 2^{n-1} linear recurrences of order m. Suppose that all the recurrences are solved in parallel by using a fast parallel recurrence solving algorithm like the one in (16) or (20) because m is expected to be "small" in practice. Then the recurrences can be solved in approximately $(2 + \log m) \log n$ time with an unlimited number of processors. So, any arbitrary linear mixed recurrence loop of order m and of size n can be executed in $O(\log n)$ time with an unlimited number of processors, assuming $m \ll n$.

Although parallelizing a general linear mixed-recurrence loop based on its binary tree representation seems less efficient than one may hope for, a majority of linear mixed recurrence loops encountered in practice are of simpler forms: either postfix-IF loops or loops having very simple linear recurrence relations of order 1. These simpler forms of linear mixed recurrence loops limit the number of possible values of predicate variable, and the loops can be parallelized with a better efficiency.

Because a postfix-IF loop does not include a linear recurrence relation, it does not require solving a set of recurrence equations to set up its binary tree representation. A postfix-IF loop of order m and of size n can be executed in $O(\log n)$ time with an unlimited number of processors by taking the first 2^m nodes at each level from its binary tree representation. Notice that all the $e_{i,u}$'s result in the same value of b_i as $e_{i,j}$ where $j = (u - 1) \bmod 2^{m-1} + 1$, because the value of each $e_{i,u}$ depends only on the values of $e_{l,t}$'s $[(i - m) \le l \le (i - 1)]$. In a postfix-IF loop of order m, there are at most 2^m nodes at each level of the tree. Because p processors can cover $p/2^m$ levels of the tree, path-selection can be done in $O[(2^m{}_n/p) \log (p/2^m)]$ time. Because it is rare for m to be greater than three, we may consider 2^m to be a constant. So parallelizing a postfix-IF loop gives a reasonable speedup of $O(p/\log p)$.

In most linear mixed recurrence loops, the linear recurrence that needs to be solved is order 1 with constant coefficients. Furthermore, the coefficients take the value of either 1 or 0. Consider a linear mixed

recurrence loop of order 1, which can be represented by the following equation:

$$x_i = \begin{cases} a_i \cdot x_{i-1} + c_i & \text{if } b_{i-1} = 1 \\ \bar{a}_i \cdot x_{i-1} + \bar{c}_i & \text{if } b_{i-1} = 0 \end{cases} \quad (2 \le i \le n)$$

where a_i, c_i, $\bar{a}_i$, and $\bar{c}_i$ are coefficients. Then we have the following three special cases:

Case I: $c_i = 0$, $\bar{c}_i = 0$, a_i are fixed for all i, and $\bar{a}_i$ are fixed for all i.
Case II: $a_i = 1$, $\bar{a}_i = 1$, c_i are fixed for all i, and $\bar{c}_i$ are fixed for all i.
Case III: Either $a_i = 0$ for all i or $\bar{a}_i = 0$ for all i (0 and 1 are integer)

Notice that all the special cases are recurrences with constant coefficients. Cases I and II are cases of constant coefficients with one of the two coefficients knocked off, and Case III has a linear recurrence from only one side of the branching.

The number of possible cases for evaluating the predicate of the branching is drastically reduced in these special cases. By the commutativity principle of multiplication (for Case I), by the commutativity principle of addition (for Case II), and by induction (for Case III) there are i possible cases of the predicate evaluation at ith iteration. This reduced number of possible cases of the predicate evaluation and the constant coefficients naturally simplify the precomputation to set up the binary tree representation and the path selection from the tree. These simple forms of first-order linear mixed recurrence loops can be parallelized with speedup proportional to $n/\log n$ with n^2 processors (see Ref. 13 for the experimental results of program parallelization using the approach described).

An Interprocedural Array Data-Flow Analysis

The dependence definition says that two statements are dependent if they access the same memory locations. Dependence analysis based on this definition is also called *memory disambiguation* (or *address-based data dependence analysis*). Memory disambiguation is an approximation to the exact dependence analysis which is based on values, as defined originally in Ref. 17.

For a dependence definition based on values, consider two statements S_i and S_j when S_i precedes S_j on the control flow path of a given program. Several types of dependence are possible (5,17).

(1) S_j is data *flow dependent* on S_i if a value of a variable used by S_j was computed by S_i.
(2) S_j is data *antidependent* on S_i if a value of a variable after being used by S_i is recomputed by S_j.
(3) S_j is *output dependent* on S_i if both compute the same variable and a value of the variable computed by S_j is to be stored after that computed by S_i.

In contrast to address-based dependence analysis, dependence analysis based on the preceding definition is called a *value-based data dependence analysis*.

Although memory disambiguation is very useful in practice, its limitations have been reported in Ref. 21. One of its limitations is array privatization (22,23), which is important for loop parallelization. As a result, many approaches to value-based dependence analysis have been proposed (24,25,26,27), as well as the one presented next.

Interprocedural Analysis. Procedural calls are frequently used inside loops, and the loops containing calls usually have more computations. Therefore, it is important to parallelize loops that contain calls. Unfor-

tunately, the pairwise dependence tests discussed previously cannot be easily extended to handle procedural calls. As a result, analyzing procedural calls has been studied widely (23,28,29,30). Procedural calls can be handled by either inlining or interprocedural analysis. Inlining replaces calls by their corresponding routines, and in general, it is expensive. Interprocedural analysis summarizes the side effects of a called routine with sets of array elements that are modified or used by routine calls, called MOD sets and USE sets, respectively. Data dependences involving routine calls can be tested by intersecting these sets. Existing approaches can be categorized according to methods of set representation. Convex regions (8) and data access descriptors (31) define sets by a system of inequalities and equalities, while bounded regular sections (28,29) use range tuples to represent sets. Even though bounded regular sections are less precise than convex regions and data access descriptors, they are much easier to implement.

A *flow-sensitive* summary approach is a summary approach in which control flow information is needed for collecting summary information. The approaches that collect only USE and MOD sets of array elements are not flow-sensitive (also called *flow-insensitive*). Furthermore, an approach is called *path-sensitive* if branching conditions are taken into account to distinguish the summary information collected for different branches. The experiment (23,32) shows that a powerful approach should be flow-sensitive and path-sensitive. Such an approach that collects sets of upwards exposed uses (*UE*) in addition to USE sets and that uses *guarded array regions* as its set representation is discussed next.

Guarded Array Regions. A guarded array region (*GAR*) contains a regular array region and a guard. An array region is a bounded regular section, denoted by $A(r_1, r_2, \ldots, r_m)$, where m is the dimension of A; each of r_i, r_2, ..., and r_m is a range in the form of $(l{:}u{:}s)$, and l, u, s are symbolic expressions. The triple $(l{:}u{:}s)$ represents all values from l to u with step s. We write (l) to mean $(l{:}u{:}s)$ if $l = u$, and $(l{:}u)$ to mean $(l{:}u{:}s)$ if $s = 1$.

A GAR is a tuple $[P, R]$ that contains a *regular array region* R and a guard P, where P is a predicate that specifies the condition under which R is accessed. If either P is false or R is $\emptyset$, we say that $[P, R]$ is $\emptyset$. For simplicity, $[P, R]$ is denoted by R if P is true (T).

For any given program segment that has a unique entry node and a unique exit node, the side effect of a program segment can be captured by modification sets (MOD sets) and upward-exposed sets (*UE* sets). Take the following segment for example:

```
DO I=2, N A(I)=A(I−1)+B(I) B(I)=C(I)+B(I) ENDDO
```

Consider the loop body first. The MOD sets and UE sets, for an arbitrary iteration i, follow:

(1) A:
(2) B:
(3) C:

(1) MOD: $A(i)$
(2) MOD: $B(i)$
(3) MOD: $\emptyset$

(1) UE: $A(i-1)$
(2) UE: $B(i)$
(3) UE: $C(i)$

For convenience, these MOD sets and UE sets are represented by MOD_i and UE_i, respectively. The subscript i indicates that the sets are for an arbitrary iteration i. Similarly, $\text{MOD}_{<i}$ and $\text{UE}_{<i}$ represent MOD and UE sets, respectively, for all iterations prior to iteration i, where $\text{MOD}_{>i}$ and $\text{UE}_{>i}$ represent MOD and

UE sets, respectively, for all iterations after iteration i. For array A, these sets are

$$\mathbf{MOD}_{<i} = A(2:i-1), \mathbf{MOD}_{>i} = A(i+1, N)$$
$$\mathbf{UE}_{>i} = \emptyset, \mathbf{UE}_{<i} = \begin{cases} A(1), & \text{if } i = 2 \\ \emptyset & \text{otherwise} \end{cases}$$

For the loop, the MOD sets and UE sets are

(1) A:
(2) B:
(3) C:

(1) MOD: $A(2{:}N)$
(2) MOD: $B(2{:}N)$
(3) MOD: $\emptyset$

(1) UE: $A(1)$
(2) UE: $B(2{:}N)$
(3) UE: $C(2{:}N)$

Operations on GARs. Three kinds of operations on GARs—namely union, intersection, and difference—are necessary for the array data-flow analysis. These operations in turn are based on union, intersection, and difference operations on regular array regions as well as logical operations on predicates. Here, we discuss only the top-level operations and refer the readers to 23 for more details. Given two GARs, $T_1 = [P_1, R_1]$ and $T_2 = [P_2, R_2]$, we describe the set operations next:

- $T_1 \cap T_2 = [P_1 \wedge P_2, R_1 \cap R_2]$
- $T_1 \cup T_2$ Two cases of union operations are the most frequent:
  - If $P_1 = P_2$, the union becomes $[P_1, R_1 \cup R_2]$
  - If $R_1 = R_2$, the result is $[P_1 \vee P_2, R_1]$
- $T_1 - T_2 = [P_1 \wedge P_2, R_1 - R_2] \cup [P_1 \wedge \bar{P}_2\, R_1]$

Because symbolic variables may appear in both arithmetic expressions and predicates, the results of these operations may not be known. To avoid the loss of accuracy resulting from this fact, these operations are handled under the following rules. For a union operation, two GARs are kept in a list when they cannot be merged together. For an intersection operation, the difference $T_1 - T_2$ is not evaluated—unless the result is a single GAR or until the last moment at which the actual result must be solved in order to finish data-dependence tests or array privatizability tests. When the difference has not yet been evaluated by these formula, it is kept (see Ref. 33 for details).

The intersection operation is needed in data dependence tests, in array privatizability tests, and in the simplification of array regions. In the process of collecting summary sets, the intersection is not used. In other words, the intersection does not affect the accuracy of summary sets; it affects final dependence tests and array

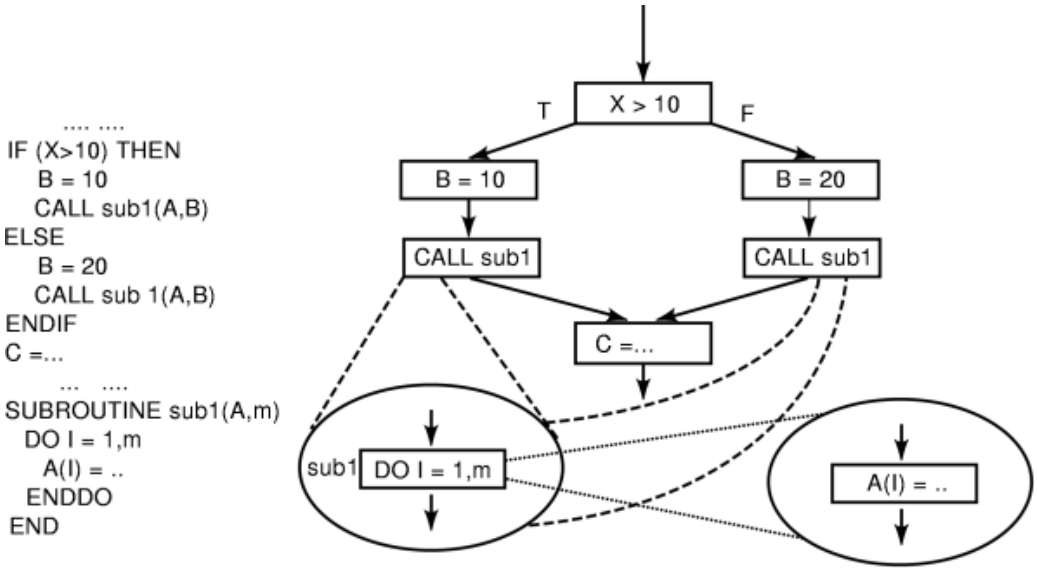

Fig. 9. Example of the HSG.

privatizability tests. When the result of an intersection is unknown, a demand-driven symbolic analysis is used to evaluate symbolic variables.

Collecting Summary Sets. By propagating the summary sets for each node over a *hierarchical supergraph* (*HSG*), the MOD and UE information can be calculated. The HSG contains three kinds of nodes—basic block nodes, loop nodes, and call nodes. An IF condition itself forms a single basic block node. A DO statement forms a loop node. A loop node is a compound node that has its attached flow subgraphs describing the control flow within the DO loop. A call statement forms a call node with its outgoing edge pointing to the entry node of the flow subgraph of the called routine. The call node also has an incoming edge from the unique exit node of the called routine. Because of the nested structures of DO loops and routines, a hierarchy is derived among the HSG nodes, with the flow subgraph at the highest level representing the main program. We assume that the program contains no recursive calls. For simplicity of presentation, we further assume that a DO loop does not contain GOTO statements that branch out the loop (therefore, it is an iterative DO loop). We also assume that the HSG contains no cycles that result from backward GOTO statements. Under these assumptions and treatment, the HSG is a hierarchical *dag* (directed acyclic graph). Figure 9 shows an HSG. Because the guards are attached to regular array regions, the calculation of the MOD information involves only union operations. The calculation of the UE information, on the other hand, requires both union and difference operations.

The algorithm for summarizing a code segment is named sum_segment. For simplicity, we consider that the algorithm is to summarize one array only. (In practice, the algorithm summarizes all arrays at the same time.) Let UE(n) and MOD(n) be the UE set and the MOD set for node n, respectively, and let UE_IN(n) and MOD_IN(n) be the UE set and MOD set for the part of the currently summarized segment that is reachable from node n, respectively. The algorithm follows:

```
sum_segment(mod, ue, G(s,e)) /* G(s,e): flow subgraph with starting node s and exist-
ing node e. */ /* mod is the mod set of G(s,e). */ /* ue is the upward exposed use set
of G(s,e). */ Step 1: Find UE(n) and MOD(n) for each node n in G(s,e). FOR each node n
in G(s,e) DO IF (n is a basic block) Summarize n; Guards in GARs are set true; ELSE IF
(n is a loop node) Let g'(s',e') be the flow subgraph of the loop body. sum_segment(m, u,
```

g′(s′,e′)); UE(n) = expand(u), MOD(n) = expand(m); ELSE IF (n is a call node) Let g′(s′,e′) be the flow subgraph of the called routine. sum_segment(m, u, g′(s′,e′)); UE(n) = map(u), MOD(n) = map(m); ENDIF ENDFOR Step 2: Propagate MOD and UE of each node backward, from e to s. Propagation follows the following flow equations and rules. MOD_IN(n) = MOD(n) ∪ ($\cup_{p \in succ(n)}$ MOD_IN(p)) UE_IN(n) = UE(n) ∪ ($\cup_{p \in succ(n)}$ UE_IN(p) − MOD(n)) (Note that succ(e) = ∅.)

If n is a basic block containing IF-condition, add the condition to the guard of each GAR in MOD_IN(n) and UE_IN(n) If any expression in the MOD_IN(n) and UE_IN(n) contains a variable that is defined within n, then that variable must be substituted by the right-hand side of the defining statement within n. If the right-hand side is too complicated, the expression is marked as unknown. If a variable is defined by a procedure or a function, we propagate information through the subgraph of this procedure or function. At the end of the propagation, we have mod = MOD_IN(s), ue = UE_IN(s)

In this algorithm, function expand( ) is used to expand summary sets for a loop body into the summary sets for the whole loop, whereas function map( ) is used to map the summary sets to the calling context. Function expand( ) in general can be complex, but it can be computed easily for most cases in practice. The mapping process may involve array reshaping. These functions are discussed in more details in the literature (23,33).

The summary sets for a basic block node can be computed easily. Figure 10 shows an example. Suppose that we want to summarize the loop body of loop S0. The simplified HSG is shown on the right-hand side of the figure, in which the details of each DO compound node are omitted for simplicity. s and e are the starting and exiting nodes, respectively. Suppose that we have the summary sets for each loop node, S2 and S5, i.e.,

(1) Loop S2:

(1) A:
(2) MOD: $A(1{:}M)$
(3) UE: ∅

(1) B:
(2) MOD: ∅
(3) UE: $B(i,1{:}M)$

(1) Loop S5:

(1) A:
(2) MOD: ∅
(3) UE: $A(1{:}M)$

(1) B:
(2) MOD: $B(i,1{:}M)$
(3) UE: ∅

The process of propagation for array A is described as follows:

MOD_IN (S5) = ∅, UE_IN(S5) = $A(1{:}M)$ MOD_IN (S4) = ∅, UE_IN(S4) = [p, $A(1{:}M)$] MOD_IN (S2) = MOD(S2) = $A(1{:}M)$, UE_IN(S2) = [p, $A(1{:}M)$] − $A(1{:}M)$ = ∅ MOD_IN (S1) = [p, $A(1{:}M)$], UE_IN(S1) = ∅

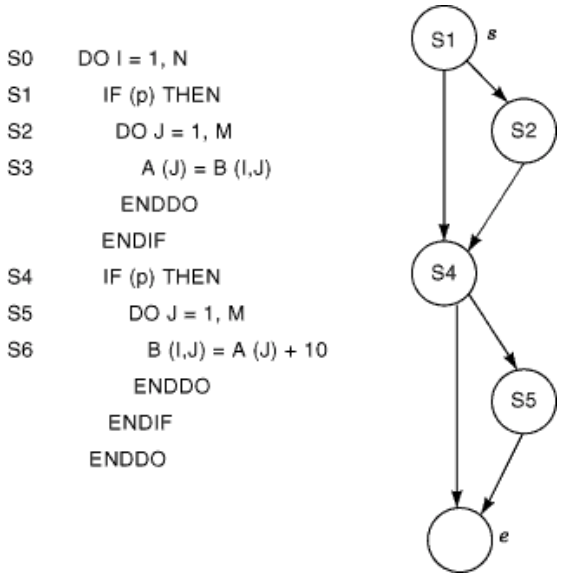

Fig. 10. A loop with a privatizable array.

The summary sets for the body of loop S1, denoted by UE_i and MOD_i, are MOD_IN(S1) and UE_IN(S1), respectively, that is,

$$\mathbf{UE}_i = \emptyset \text{ and } \mathbf{MOD}_i = [p, A(\mathbf{1}{:}M)]$$

Similarly, the summary sets for array B are

$$\mathbf{UE}_i = [p, B(i, \mathbf{1}{:}M)] \text{ and } \mathbf{MOD}_i = [p, B(i, \mathbf{1}{:}M)]$$

Array Privatization and Loop Parallelization. An array A is a privatization candidate in a loop L if its elements are overwritten in different iterations of L (see Ref. 22). Such a candidacy can be established by examining the array subscripts: if the subscripts of array A do not contain any induction variables of L, then A is a candidate. A privatization candidate is privatizable if there exist no loop-carried flow dependences in L. For an array A in a loop L with an index I, if $MOD_{<i} \cap UE_i = \emptyset$, then there exists no flow dependence carried by loop L. In Fig. 10, for example, array A is obviously a privatization candidate. Because $UE_i = \emptyset$, we have $MOD_{<i} \cap UE_i = \emptyset$. So A is privatizable within loop S0.

The essence of loop parallelization is to prove the absence of loop-carried dependences. For a given DO loop L with index I, the existence of different types of loop-carried dependences can be detected in the following order:

(1) *Loop-carried flow dependences* exist if and only if $UE_i \cap MOD_{<i} \neq \emptyset$.

(2) *Loop-carried output dependences* exist if and only if $MOD_i \cap (MOD_{<i} \cup MOD_{>I}) \neq \emptyset$.

(3) *Loop-carried antidependences* exist if and only if $UE_i \cap MOD_{>i} \neq \emptyset$. This formula is valid in the absence of loop-carried output dependences. It is applied only after our algorithm successfully proves the absence of loop-carried flow and loop-carried output dependences in steps 1 and 2. (If loop-carried antidependences are

```
DOALL I=1, N
  Private A
  If (p) THEN
    DO J=1, M
      A(J) = B(I,J)
    ENDDO
  ENDIF
  IF (p) THEN
    DO J=1, M
      B(I,J) = A(J) + 10
    ENDDO
  ENDIF
ENDDOALL
```

Fig. 11. A parallel version of the loop in Fig. 10.

considered separately, they should be detected using DE_i instead of UE_i in the preceding formula, where DE_i is the *downwards exposed* use set of iteration i.)

It is not difficult to show that array B in Fig. 10 does not cause any loop-carried dependence. Actually, we have $\mathrm{MOD}_{<i} = [p, B(1{:}i-1, 1{:}M)]$ and $\mathrm{MOD}_{>i} = [p, B(i+1{:}N, 1{:}M)]$. The intersection of $\mathrm{UE}_i \cap \mathrm{MOD}_{<i}$ is empty, that is,

$$\mathbf{UE}_i \cap \mathbf{MOD}_{<i} = [p, B(\mathbf{1}:i-\mathbf{1}, \mathbf{1}:M)] \cap [p, B(i, \mathbf{1}:M)] = \emptyset$$

The intersections of $\mathrm{MOD}_i \cap (\mathrm{MOD}_{<i} \cup \mathrm{MOD}_{>i})$ and of $\mathrm{UE}_i \cap \mathrm{MOD}_{>i}$ are also empty. Therefore, loop S0 is parallelizable. The parallel version of the loop is shown in Fig. 11.

Further Reading

Traditional dependence analysis for loop parallelization is discussed in detail in Refs. 1, 9, and 34. More details on parallelizing loops with recurrence can be found in Ref. 12. The array data-flow analyses that do not handle procedural calls are proposed in Refs. 25,26,27 and 35,36,37, whereas the summary array data-flow analyses that can handle procedural calls are proposed in Refs. 22, 33, and 38,39,40.

BIBLIOGRAPHY

1. M. Wolfe *High Performance Compilers for Parallel Computing*, Redwood City, CA: Addison-Wesley, 1996.
2. U. Banerjee *Loop Transformations for Restructuring Compilers: The Foundations*, Norwell, MA: Kluwer Academic, 1993.
3. H. Zima B. Chapman *Supercompilers for Parallel and Vector Computers*, Reading, MA: Addison-Wesley, 1991.
4. R. Kuhn *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees.* Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, Feb 1980.
5. M. J. Wolfe *Optimizing Super Compilers for Supercomputers*, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 1982.

6. M. J. Wolfe U. Banerjee Data dependence and its application to parallel processing, *Int. J. Parallel Programming*, **16** (2): 137–178, 1987.
7. U. Banerjee *Dependence Analysis for Supercomputing*, Norwell, MA: Kluwer Academic, 1988.
8. R. Triolet *Interprocedural analysis for program restructuring with Parafrase, Technical Report CSRD Rpt. No. 538*, Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign, December 1985.
9. D. E. Maydan *Accurate Analysis of Array References*, Ph.D. thesis, Stanford University, October 1992.
10. W. Pugh A practical algorithm for exact array dependence analysis. *Commun. ACM*, **35** (8): 1992.
11. C. Eisenbeis J.-C. Sogno A general algorithm for data dependence analysis, *6th ACM Int. Conf. Supercomput.*, July 1992.
12. M. Wolfe *Optimizing Supercompilers for Supercomputers*, Cambridge, MA: MIT Press, 1989.
13. G. Lee C. Kruskal D. J. Kuck An empirical study of automatic restructuring of nonnumerical programs for parallel processors, *IEEE Trans. Comput.*, **C-34**: 927–933, 1985.
14. U. Banerjee D. Gajski Fast evaluation of loops with if statement, *IEEE Trans. Comput.*, **C-33**: 1030–1033, 1984.
15. R. A. Towle *Control and Data Dependence for Program Transformations*, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 1976.
16. S. Chen D. Kuck A. Sameh Practical band triangular system solvers, *ACM Trans. Math. Software*, **4** (3): 270–277, 1978.
17. D. J. Kuck *The Structure of Computers and Computations*, Vol. 1, New York: Wiley, 1978.
18. I. Munro M. Paterson Optimal algorithms for parallel polynomial evaluation, *J. Comput. Syst. Sci.*, **7**: 189–198, 1973.
19. R. E. Ladner M. J. Fischer Parallel prefix computation, *J. ACM*, **27** (4): 831–848, 1980.
20. A. Sameh R. Brent Solving triangular systems on a parallel computer, *SIAM J. Numer. Anal.*, **14** (6): 1101–1113, 1977.
21. W. Blume R. Eigenman Performace analysis of parallelizing compilers on the Perfect benchmarks programs, *IEEE Trans. Parallel Distrib. Syst.*, **3**: 643–656, 1992.
22. Z. Li Array privatization for parallel execution of loops, *ACM Int. Conf. Supercomput.*, pp. 313–322, July 1992.

23. J. Gu Z. Li G. Lee Symbolic array dataflow analysis for array privatization and program parallelization, *Supercomputing '95*, December 1995.
24. T. Brandes The importance of direct dependences for automatic parallelization, *ACM Int. Conf. Supercomput.*, July 1988.
25. P. Feautrier Dataflow analysis of array and scalar references, *Int. J. Parallel Programming*, **2** (1): 23–53, 1991.
26. W. Pugh D. Wonnacott An exact method for analysis of value-based array data dependences, *6th Annu. Workshop Programming Languages Compilers Parallel Comput.*, Portland, OR, in *Lecture Notes in Computer Science 768*, Berlin: Springer-Verlag, August 1993.
27. T. Gross P. Steenkiste Structured dataflow analysis for arrays and its use in an optimizing compiler, *Software—Practice Experience*, **20** (2): 133–155, 1990.
28. D. Callahan K. Kennedy Analysis of interprocedural side effects in a parallel programming environment, *ACM SIGPLAN '86 Symp. Compiler Construction*, June 1986, pp. 162–175.
29. P. Havlak K. Kennedy An implementation of interprocedural bounded regular section analysis, *IEEE Trans. Parallel Distrib. Syst.*, **2**: 350–360, 1991.
30. B. Creusillet F. Irigoin Interprocedural array region analyses, *Proc. 8th Workshop Languages Compilers Parallel Comput., No. 1033*, in *Lecture Notes in Computer Science*, Berlin: Springer-Verlag, August 1995, pp. 46–60.
31. V. Balasundaram A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor, *J. Parallel Distributed Comput.*, **9**: 154–170, 1990.
32. R. Eigenmann J. Hoeflinger D. Padua *On the automatic parallelization of the Perfect benchmarks, Technical Report TR 1392, CSRD*, University of Illinois, Urbana-Champaign, November 1994.
33. J. Gu Z. Li G. Lee Experience with efficient array data flow analysis for array privatization, *6th ACM SIGPLAN Symp. Principles Practice Parallel Programming*, June 1997, pp. 157–167.
34. U. Banerjee *Dependence Analysis*, Norwell, MA: Kluwer Academic, 1997.
35. E. Duesterwald R. Gupta M. L. Soffa A practical data flow framework for array reference analysis and its use in optimizations, *ACM SIGPLAN '93 Conf. Programming Language Design Implementation*, June 1993, pp. 68–77.
36. J. F. Collard D. Barthou P. Feautrier Fuzzy array dataflow analysis, *ACM SIGPLAN Symp. Principles Practice Parallel Programming*, June 1995.
37. V. Maslov Lazy array data-flow dependence analysis, *Proc. Annu. ACM Symp. Principles Programming Languages*, Jan. 1994, pp. 311–325.
38. P. Tu D. Padua Automatic array privatization, *Proc. 6th Workshop Languages Compilers Parallel Comput.*, August 1993, pp. 500–521.
39. M. W. Hall *et al.* Interprocedural analysis for parallelization, *Proc. 8th Workshop Languages Compilers Parallel Comput., No. 1033*, in *Lecture Notes in Computer Science*, Berlin: Springer-Verlag, August 1995, pp. 61–80.
40. B. Creusillet *Array Region Analysis and Applications*, Ph.D. thesis, École des Mines de Paris/CRI, December 1996.

GYUNGHO LEE
University of Texas—San Antonio
JUNJIE GU
University of Minnesota