Interprocedural Induction Variable Analysis

Peiyi Tang
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR 72204

Pen-Chung Yew
Department of Computer Science and Engineering
University of Minnesota
Minneaplolis, MN 55455

Abstract[1]

Induction variable analysis is an important part of the symbolic analysis in parallelizing compilers. Induction variables can be formed by `for` *or* `DO` *loops within procedures or loops of recursive procedure calls. This paper presents an algorithm to find induction variables in formal parameters of procedures caused by recursive procedure calls. The compile-time knowledge of induction variables in formal parameters is essential to summarize array sections to be used for data dependence test and parallelization.*

1. Introduction

Induction variable analysis is to find the scalar variables in programs whose values can be expressed in linear forms. Finding induction variables enables parallelizing compilers to form accurate array sections [1, 2, 3] accessed in loops for loop parallelization.

Induction variables are always formed by loops in programs. Much work has been done to discover induction variables in local variables formed by explicit loops such as `for` or `DO` loops [4, 5]. However, induction variables can also be formed by loops of recursive procedure calls. For instance, in the recursive procedure `x` in Figure 1(b), formal parameter `k` is an induction variable $\{k_0 + 2i \mid 0 \leq i \leq \lfloor n/2 \rfloor - 2\}$ and it is formed by the loop of recursive call of `x` to itself. At the same time, parameters `i` and `n` are invariant with respect to that loop. As a result, the section of array `b` modified by the assignment `b(i,k) = ...` is the even elements of the row i: $b(i, k_0), b(i, k_0 + 2), \cdots$. With this knowledge, the parallelizing compiler would know that there are no data dependences between statements `s0` and `s1` carried by loop `i` in the program Figure 1(a), because the data elements accessed in statement `s0` by loop `j` are the odd elements of the rows of array `a`. (Notice that array `a` is aliased with array `b` by the call statement at `s2`.) Therefore, the parallelizing compiler can parallelize loop `i`. Without this knowledge, the compiler would assume that all the elements of row i of array `a` be modified by the procedure call at `s1` and there would be a data dependence cycle between `s0` and `s1` which would prevent the parallelization of loop `i`.

```
real a(n,n)
do i = 1, n
    do j = 1, n, 2
s0:   a(i,j) = a(i-1,j) + ...
    enddo
s1: call x(a,i,2,n)
enddo
```

(a) loop nest

```
subroutine x(b,i,k,n)
  real b(n,n)
  b(i,k) = ...
  if (k+1 < n) then
    call x(b,i,k+2,n)
  endif
end
```

(a) recursive procedure

Figure 1. Interprocedural Induction Variables

The induction variables in procedure parameters formed by the loops of recursive procedure calls or returns are called *interprocedural induction variables*.

Previous research on induction variable analysis [4, 5, 6] is primarily concerned with induction variables formed by explicit loops within procedures. Although [6] mentioned interprocedural induction variable analysis, but the induction variables targeted are still formed by explicit loops. To the best of our knowledge, there is no previous work on discovery and analysis of interprocedural induction variables defined above.

In this paper, we present an algorithm to discover and

[1] The work was supported in part by the U.S. National Science Foundation under Grants EIA-9971666 and MIP-9610379 and a grant from the Intel Corporation.

IEEE COMPUTER SOCIETY

analyze interprocedural induction variables in parameters of procedures.

The loops of recursive procedure calls or returns are *implicit*, because they do not exist in the abstract syntax trees of the program. More importantly, the structures of these loops are quite different from those of ordinary *explicit* `for` or `DO` loops. While the basic technique of detecting induction variables remains the same as the *intra*procedural induction variable analysis, the *inter*procedural induction variables analysis first needs to recover, identify and analyze these implicit loops. The contributions of this paper are: (1) the techniques to identify and analyze the unique structure of implicit loops of recursive call and returns and (2) the complete algorithm to identify and analyze interprocedural induction variables.

We use FORTRAN for the base program model in this paper, but allow recursive procedure calls. We will concentrate on scalar parameters of procedures for the induction variable analysis and assume call-by-reference for all parameters. To simplify presentation, we do not consider global variables.

The purpose of the analysis in this paper is to classify all the scalar parameters of procedures to be either (1) loop invariant variables with respect to the loop of recursive procedure calls or returns, or (2) induction variables with respect to the corresponding loop, or (3) complex variables whose values cannot be determined as loop invariants or induction variables using our method. Loop invariants can be regarded as a special case of induction variables with the induction step to be 0.

The rest of the paper is organized as follows. Section 2 describes the extended full program representation graph we use in the analysis. Section 3 describes the loop structure of recursive calls and returns. Section 4 presents the algorithm to find interprocedural induction variables. Section 5 concludes the paper with related work and concluding remarks.

2. Extended Full Program Representation (EFPR) Graph

In order to analyze *inter*procedural induction variables, the compiler first needs to recover and identify the implicit loops of recursive calls and returns. We extended the full program representation graph by Agrawal el al [7] for this purpose.

Suppose there are n procedures in the program. Each procedure has a *start* node and a *return* node in the EFPR graph. The start node and the return node of procedure $i(1 \leq i \leq n)$ are denoted s_i and r_i, respectively. Let S and R be the sets of start nodes and return nodes of all procedures, respectively, i.e. $S = \cup_{i=1}^{n} \{s_i\}$ and $R = \cup_{i=1}^{n} \{r_i\}$. Let B_i be the set of branch nodes in the control flow graph of procedure i and $B = \cup_{i=1}^{n} B_i$. The extended full program representation (EFPR) graph is a directed graph $G = (V, E)$ whose node set is $V = S \cup R \cup B$. Before we define the edge set E, let us define set A_i for procedure i to be $A_i = B_i \cup \{entry_i\} \cup \{exit_i\}$, where $entry_i$ and $exit_i$ are the entry node and the exit node of the control flow graph of procedure i. The edge set E of G is defined as follows:

1. If procedure k calls procedure i at call site cs, and there is a control flow path from a node $a \in A_k$ of procedure k to cs which does not contain any other call statements or any other branch nodes in B_k, there is an edge $(a, s_i) \in E$.

2. If procedure k calls procedure i at call site cs, and there is a control flow path from cs to a node $a \in A_k$ of procedure k which does not contain any other call statements or any other branch nodes in B_k, there is an edge $(r_i, a) \in E$.

3. If procedure k calls procedures i and j at call sites cs_1 and cs_1, respectively, and there is a control flow path from cs_1 to cs_2 which does not contain any other call statements or any other branch nodes in B_k, there is an edge $(r_i, s_j) \in E$.

4. If there is a control flow path from node $a_1 \in A_k$ to another node $a_2 \in A_k$ in the control flow graph of procedure k which does not contain any other call statements or any other branch nodes in B_k, there is an edge $(a_1, a_2) \in E$.

Figure 4(a) shows the EFPR graph for the program in Figure 3. We use rectangles to represent branch nodes in EFPR graphs in this paper.

The EFPR graph captures all the control flows of the whole program. It includes both the explicit `for` or `DO` loops in all procedures and the implicit loops of recursive calls and returns.

3. Loop Structure of Recursive Calls and Returns

In general, the implicit loops of recursive calls and returns are not as well-structured as the explicit `for` or `DO` loops. For example, the explicit loops and implicit loops may be overlapped rather than nested with each other and, thus, the EFPR graph may be irreducible. A procedure may be called at more two call sites and its parameters cannot be induction variables.

In order to form interprocedural induction variables, loops of recursive calls and returns have to conform to certain structure. The compiler needs to check the loop structure of recursive calls and returns before it can proceed to

IEEE COMPUTER SOCIETY

the induction variable analysis. This can be done by checking first the call graph and then the EFPR graph of the program. The details of these checkings can be found in the technical report version of this paper [8] and are omitted here.

As in the *intra*procedural induction variable analysis, the analysis for *inter*procedural induction variables starts with the innermost loop. After the checkings of the call graph and the EFPR graph mentioned above, the structure of a typical innermost loop of recursive calls is illustrated in Figure 2, where procedures $0, 1, \cdots, n-1$ form a loop of recursive calls and s_0 is the loop header. To simplify discussion, we assume that there is only one branch node, b_i, between $s_{i-1 \bmod n}$ and s_i, $0 \leq i \leq n-1$. A dotted arrow in Figure 2 represents a *path* in the EFPR graph. It is possible that there are multiple paths from b_i to $r_{i-1 \bmod n}$ due to the possible branch nodes between them. For the same reason, there may be multiple paths from r_i to $r_{i-1 \bmod n}$. The cycles from r_{n-1} down to r_0 and back to r_{n-1} formed by the dotted arrows are the possible control flows of the returns of the corresponding recursive calls. Each of these cycles is called a *dual loop* of the loop of recursive calls.

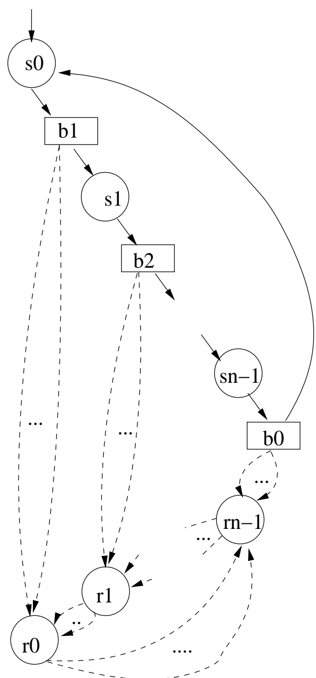


Figure 2. Structure of a typical innermost loop of recursive calls and its dual loop

The precise definition of innermost loop of recursive calls and its dual loops is as follows:

Definition 1 (Innermost Loop of Recursive Calls) *The innermost loop of recursive calls is a natural loop with header s_0 in the EFPR graph, $s_0 \rightarrow \cdots \rightarrow s_1 \cdots s_{n-1} \rightarrow \cdots \rightarrow s_0$, such that (1) none of $s_1, \cdots, s_{n-1}$ is the header of another natural loop in the EFPR graph and (2) none of paths from r_j to $r_{j-1 \bmod n}$, $(j = 0, \cdots, n-1)$ contains a start node with more than one incoming edge and (3) any path from a branch node between s_i and $s_{i+1 \bmod n}$ to the return node r_i does not contain a start node with more than one incoming edge. Each of the cycles in the EFPR graph $r_0 \rightarrow r_{n-1} \rightarrow \cdots \rightarrow r_1 \rightarrow r_0$ is called a dual loop of the the loop of recursive calls.*

4. Algorithm for Interprocedural Induction Variable Analysis

The algorithm for interprocedural induction variable analysis is as follows:

while (there is an innermost loop of recursive calls in EFPR graph) **do**

1. Identify the innermost loop of recursive calls $s_0 \rightarrow \cdots \rightarrow s_1 \cdots s_{n-1} \rightarrow \cdots \rightarrow s_0$ as defined in Definition 1 and illustrated in Figure 2.

2. Construct the interprocedural factored use-def (IFUD) graph of the procedures $0, 1, \cdots, n-1$. Apply the modified Tarjan's algorithm [4, 5] to find loop invariant, induction and complex variables in the input parameters of the procedures.

3. Calculate the trip count and the break point of the loop of recursive calls. Represent the induction variables of input parameters using the basic induction variable and the trip count of the loop.

4. Check the EFPR graph to see if there is only one path from r_j to $r_{j-1 \bmod n}$ for all $j = 0, \cdots, n-1$. If so, continue to find induction variables in the output parameters of the procedures caused by the dual loop as follows:

   (a) Check if there is only one path in the EFPR graph from the break point to the return node of the procedure to which it belongs. If there are multiple paths, go to 5; otherwise continue with 4(b).

   (b) Use the IFUD graph to find the output parameters which are constants or dependent only on the input parameters which are induction variables or loop invariants. These output parameters will have constant initial values. Mark the output parameters which do not have constant initial values as complex variables.

IEEE COMPUTER SOCIETY

(c) Apply the modified Tarjan's algorithm to find loop invariant, induction and complex variables in the output parameters. Represent the induction variables of output parameters using the same basic loop induction variable and the trip count as the loop of recursive calls.

5. Coalesce both the innermost loop and its dual loop in the EFPR.

We next describe each step of the algorithm in detail. We also use the program in Figure 3 as the working example to illustrate the algorithm.

```
program main
   call X(1,9,1)
end

subroutine X(A,B,U)
     A=A+1
c1: if (A<B) then
      call Y(A,B,U)
     endif
end
subroutine Y(C,D,V)
     D=D-2
     V=V+D
c2: if (C<D) then
       call X(C,D,V)
       C=C-2
     endif
end
```

Figure 3. Program of Working Example

4.1. Finding Innermost loop of Recursive Calls

The innermost loop of recursive calls is defined in Definition 1. The algorithm of finding natural loops described in [9] can be used to find all the natural loops in the EFPR graph. Using Definition 1, the compiler is able to find the innermost loop of recursive calls and all its dual loops as illustrated in Figure 2.

4.2. Detecting Interprocedural Induction Variables

The compiler needs the data flow information among the parameters of the procedures. This information is embodied in the interprocedural factored use-def (IFUD) graph defined as follows.

Each call-by-reference parameter, X, is regarded as both input and output variables, called *input* and *output* parameters, respectively. At the entry node of the procedure, the value of the input parameter is one of the values passed at the call sites in the program. Thus, the input parameter denoted X is modeled by a ϕ-term, $\phi(X)$, at the entry of the procedure with the passed values as its sources. On the other hand, the value of the output parameter denoted X' at the exit node of the procedure is one of the values reaching from within the procedure. Thus, it is also modeled by a ϕ-term, $\phi(X')$, at the exit node. This ϕ-term represents the value to be passed to whatever the actual parameter bound at a call site.

The IFUD graph is obtained by merging the factored use-def graphs [4, 5] of the procedures through these input and output ϕ-terms of the formal parameters. The IFUD graph of the working example is shown in Figure 4(b).

Once the IFUD graph is established, the compiler applies the modified Tarjan's algorithm [4, 5] to find all the induction variables and loop invariants in the input parameters. For the working example, the compiler can find that parameters A and C as well as B and D, are induction variables with induction steps 1 and -2, respectively. Parameters U and V are complex variables.

4.3. Trip Count and Break Point

To simplify presentation, we assume that there is at most one branch node between successive start nodes of the procedures in the innermost loop of recursive calls.

Given an innermost loop of recursive calls $s_0 \rightarrow \cdots \rightarrow s_1 \cdots s_{n-1} \rightarrow \cdots \rightarrow s_0$, the condition for the branch node, b_i, between $s_{i-1 \bmod n}$ and s_i to take the control to s_i is denoted C_i. If there is no such branch node b_i between $s_{i-1 \bmod n}$ and s_i, C_i is a constant logic TRUE.

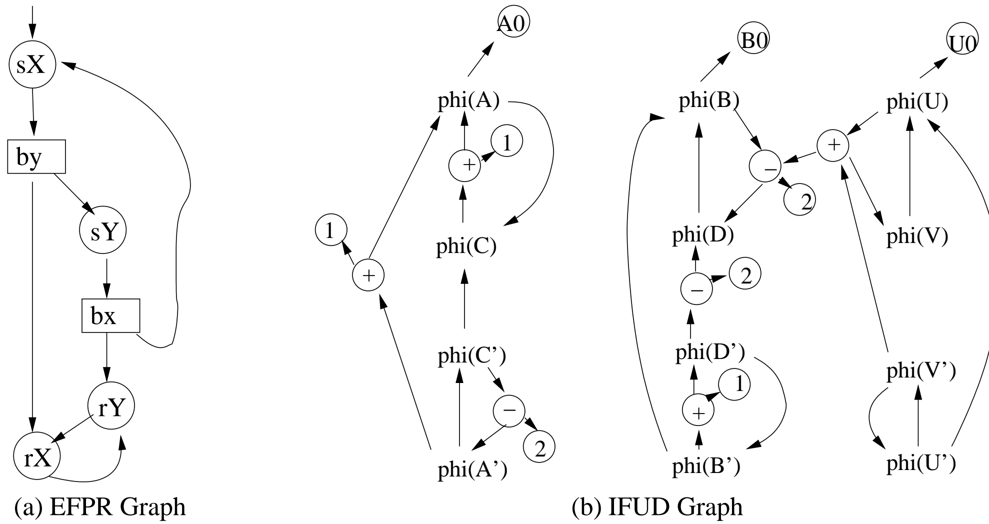
The *trip count* of the loop, denoted t_c, is the number of times the node s_0 is visited.

The condition to make a full cycle from s_0 to itself is obviously $C_1 \wedge \cdots \wedge C_{n-1} \wedge C_0$. Each of the conditions $C_0, \cdots, C_{n-1}$ can be expressed in terms of the input parameters of the procedure to which it belongs. These input parameters have already been marked as loop invariants, induction or complex variables in step 2 of the algorithm. The expression of any condition should not contain any complex variable; otherwise, the trip count of the loop should be regarded as indeterministic. These input parameters are then replaced by the linear forms of the basic loop induction variable k and each condition becomes an inequality in terms of k, denoted $C_i(k)$ $(0 \leq i \leq n-1)$.

The basic loop induction variable k is a non-negative integer starting from 0 and is incremented each time s_0 is revisited. Therefore, $C_0(k) \wedge \cdots \wedge C_{n-1}(k) \wedge (k \geq 0)$ gives the condition for the full trip of the loop in terms of k. This is an integer linear programming system of single variable. If it has no solution, then there is no full trip in the loop and s_0 is visited only once, giving $t_c = 1$. If it has solutions, we seek the maximum k satisfying the system. Let $k_{max} \geq 0$

1087-4089/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY



(a) EFPR Graph (b) IFUD Graph

Figure 4. EFPR and IFUD graphs of the working example

be the maximum integer such that $C_0(k) \wedge \cdots \wedge C_{n-1}(k)$ is true for all integers k such that $0 \leq k \leq k_{max}$. Then, there are $k_{max}+1$ full trips and the trip count t_c equals to $k_{max}+2$. The basic induction variable k of the loop takes the values $0, 1, \cdots, k_{max}, k_{max}+1$. Hence, we have

$$t_c = \begin{cases} 1 & \text{if system has no solution} \\ k_{max}+2 & \text{otherwise} \end{cases}$$

and the basic induction variable k satisfies $0 \leq k \leq t_c - 1$.

In our working example, the linear system is $C_X(k) \equiv 3k+1 < B_0 - A_0, C_Y(k) \equiv 3k+3 < B_0 - A_0$, and $k \geq 0$ and the largest k satisfying it is $k_{max} = \lfloor \frac{B_0 - A_0 - 3}{3} \rfloor = \lfloor \frac{9-1-3}{3} \rfloor = 1$.

Putting it all together, the four induction variables in the input parameters now can be expressed as follows:

$$\begin{aligned} A(k) &= \{A_0 + k \mid 0 \leq k \leq k_{max}+1\} \\ B(k) &= \{B_0 - 2k \mid 0 \leq k \leq k_{max}+1\} \\ C(k) &= \{A_0 + k + 1 \mid 0 \leq k \leq k_{max}+1\} \\ D(k) &= \{B_0 - 2k \mid 0 \leq k \leq k_{max}+1\} \end{aligned}$$

with $k_{\text{max}} = 1$.

According to the definition of k_{max} above, the last trip carrying $k_{max}+1$ is a partial one. The *break point* of the loop is the first branch node b_i $(i = 1, 2, \cdots, n-1, 0)$ such that its corresponding condition $C_i(k_{max}+1)$ is false. That is, b_i is the break point if $C_j(k_{max}+1)$ are true for all $1 \leq j < i$ and $C_i(k_{max}+1)$ is false or b_0 is the break point if $C_j(k_{max}+1)$ are true for all $1 \leq j \leq n-1$ and $C_0(k_{max}+1)$ is false.

In our working example, we have $C_X(2) \equiv 3 \cdot 2 + 1 < 9 - 1$ is true, and $C_Y(2) \equiv 3 \cdot 2 + 3 < 9 - 1$ is false. The break point is b_Y.

4.4. Induction Variables of Dual Loop

At step 4 of the algorithm, the compiler tries to find possible induction variables in the output parameters formed by the dual loop.

The compiler first tests if there is only one path from from r_j to $r_{j-1 \bmod n}$ for all $j = 0, \cdots, n-1$. If not, there are multiple dual loops and, as a consequence, none of the output parameters can be an induction variable. The entire step 4 should exit and all the output parameters should be marked as complex variables. Otherwise, the compiler proceeds with the following sub-steps:

4(a): Checking Paths from Break Point to Return Node

Assume that the break point of the loop of recursive calls shown in Figure 2 is b_j. This step is to check if there is a single path from b_j to $r_{j-1 \bmod n}$ in the EFPR graph. If there are multiple paths, the initial values for output parameters of procedure $r_{j-1 \bmod n}$ cannot be determined. There will be no closed-form linear expressions for induction variables in the output parameters and the algorithm gives up the entire step 4 and jumps to step 5.

4(b): Determining Initial Values of Output Parameters

At this step, the compiler follows factored use-def chains of procedure procedure $r_{j-1 \bmod n}$ to find a single expression in terms of its input parameters for each output parameter. If the expression contains a complex input parameter, the output parameter does not have a constant initial value and should be marked as a complex variable.

IEEE COMPUTER SOCIETY

In our working example, the initial values of output parameters C' and D', denoted C'_0 and D'_0, are $C'_0 = C(k_{max}+1)$ and $D'_0 = D(k_{max}+1) - 2$, respectively.

4(c): Detecting Induction Variables in Output Parameters

This step is the similar to step 2 of the algorithm. The compiler applies the modified Tarjan's algorithm to find induction variables or loop invariants in all the output parameters.

In our working example, the result is that both C' and D' are induction variables expressed as $C'(k') = C'_0 - 2k'$ and $D'(k') = D'_0 + k'$. Here, k' is the basic loop induction variable of the dual loop.

The *trip count of the dual loop* is defined as the number of times r_0 is visited and it is the same as that of the loop of recursive calls, t_c. The relationship among k, k', and t_c, is summarized in Theorem 1, which is easy to prove.

Theorem 1 *The trip count of of the dual loop is the same as the innermost loop of recursive calls, t_c. The basic loop induction variable of the dual loop is an integer variable k' such that $0 \leq k' \leq k_{max}+1 = t_c - 1$ and $k + k' = k_{max}+1$ hold.*

In our working example, the initial values of output parameters A' and B' can be obtained by following the IFUD chains corresponding to the path (r_Y, r_X). They are: $A'_0 = C'_0$ and $B'_0 = D'_0 + 1$. The values A' and B' then can be expressed as $A'(k') = A'_0 - 2k'$ and $B'(k') = B'_0 + k'$. The compiler then converts the expressions to use k instead of k'. The result is as follows:

$$
\begin{aligned}
C'(k) &= \{A_0 - k_{max} + 2k \mid 0 \leq k \leq k_{max}+1\} \\
D'(k) &= \{B_0 - k_{max} - 3 - k \mid 0 \leq k \leq k_{max}+1\} \\
A'(k) &= \{A_0 - k_{max} + 2k \mid 0 \leq k \leq k_{max}+1\} \\
B'(k) &= \{B_0 - k_{max} - 2 - k \mid 0 \leq k \leq k_{max}+1\}
\end{aligned}
$$

4.5. Loop Coalescing

This step 5 is to coalesce the entire the innermost loop of recursive calls and replace it with a new procedure to represent its effect. The EFPR and IFUD graphs will change accordingly and the algorithm continues with the next level innermost loop of recursive calls. The details of the technique can be found in [8] and are omitted here.

5. Related Work and Conclusion

This work is related to [1] in which a simple algorithm to find loop invariant formal parameters with respect to loops of recursive procedure calls (called *recursively invariant parameters* there) is described. Induction variables in formal parameters would be regarded as complex variables (called *recursively variant parameters* there) by that algorithm. The work in this paper goes beyond loop invariant parameters and distinguishes induction variables of both input and output formal parameters from complex formal parameters.

We have presented an algorithm to discover interprocedural induction variables in procedure parameters formed by loops of recursive calls or returns. We extended the full program representation graph [7] and the factored use-def graph [5] to be used as the framework for the analysis in this paper.

References

[1] Zhiyuan Li and Pen-Chung Yew. Interprocedural analysis for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing, Vol. II*, pages 225–244, August 1988.

[2] Peiyi Tang. Exact side effects for interprocedural dependence analysis. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, pages 137–146, Tokyo, Japan, July 1993.

[3] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[4] Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 162–174, June 1992.

[5] Michael Wolfe. *High Peformance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[6] M. R. Haghighat. *Symbolic Analysis for Parallelizing Compilers*. Kluwer Academic Pulblishers, 1995.

[7] Gagan Agrawal, Joel Salts, and Raja Das. Interprocedural partial redundancy elimination and its applications to distributed memory compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–269, June 1995.

[8] Peiyi Tang and Pen-Chung Yew. Interprocedural induction variable analysis. Technical Report 02-010, Dept of Computer Science and Engineering, University of Minnesota, 2002.

[9] Afred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.

1087-4089/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY