International Journal of Foundations of Computer Science
© World Scientific Publishing Company

INTERPROCEDURAL INDUCTION VARIABLE ANALYSIS*

PEIYI TANG
Department of Computer Science
University of Arkansas at Little Rock
2801 South University Avenue
Little Rock, Arkansas 72204
United States of America

and

PEN-CHUNG YEW
Department of Computer Science and Engineering
University of Minnesota
4-192 EE/CSci Building
200 Union Street SE
Minneapolis, Minnesota 55455
United States of America

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

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.

Keywords: Interprocedural Induction Variables; Recursive Procedure Call; Call graphs; Induction Variable Analysis; Extended Full Program Representation (EFPR) Graphs; Interprocedural Factored Use-Def (IFUD) Graphs.

1. Introduction

Induction variable analysis is an important part of the symbolic analysis in parallelizing compilers. Its purpose is to find the scalar variables in programs whose values can be expressed in linear forms. Induction variables can appear in array subscripts. Finding induction variables enables parallelizing compilers to form accurate array sections [1, 2, 3, 4] accessed in loops. The accurate array sections

*The work was supported in part by the U.S. National Science Foundation under Grants EIA-9971666, MIP-9610379 and CCR-0105574 and a grant from the Intel Corporation.

allow the data dependence analysis to discover more loop parallelism for parallel execution.

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 [5, 6]. However, induction variables can also be formed by loops of recursive procedure calls. For instance, in the recursive procedure `x` in Fig. 1(b), formal parameter `k` is an induction variable $\{2 + 2i \mid 0 \leq i \leq \lfloor \mathtt{n}/2 \rfloor - 1\}$ 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 every other elements of row i: $b(i, 2), b(i, 4), \cdots$. At the call site `s1` in the loop nest of Fig. 1(a), the section of array `a` modified by the call statement is the even elements of row i: $a(i, 2), a(i, 4), \cdots$.

```
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
```

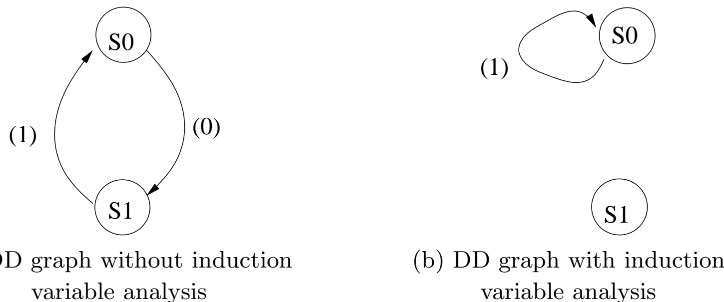
(a) Loop nest

```
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
```

(b) Recursive procedure

Fig. 1. Interprocedural induction variables

Without this knowledge of the induction variable `k`, the compiler would have to assume that `k` can take any value within its range and the section of array `a` modified by the call statement at `s1` is the entire row i, i.e. $a(i, 1 : n : 1)$. The data dependence graph[a] of the loop in Fig. 1(a) would be as shown in Fig. 2(a), where the dependence cycle prevents loop `i` from being distributed.



(a) DD graph without induction variable analysis

(b) DD graph with induction variable analysis

Fig. 2. Data dependence graphs for parallelization

But, with the knowledge of the induction variable `k`, the array section of `a` modified by the call statement is $a(i, 2 : n : 2)$. Because the array sections of `a` modified and referenced by loop `j` are $a(i, 1 : n : 2)$ and $a(i - 1, 1 : n : 2)$, respectively, there are no data dependences between statements `s0` and `s1` carried by loop `i`. Fig. 2(b) shows the sharper data dependence graph for the loop in

[a] The labels of the arrows are dependence distance vectors.

Fig. 1(a). Now loop `i` can be distributed over loop `j` and the call statement to form a parallel section. Loop `j` can be exchanged with loop `i` after the loop distribution. Then the compiler can parallelize the whole program as shown in Fig. 3.

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 [7, 8, 9, 10, 5, 6, 11] is primarily concerned with induction variables formed by explicit loops within procedures. Although [10, 11] 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.

```
real a(n,n)
par section
  doall j = 1, n, 2
    do i = 1, n
     s0:  a(i,j) = a(i-1,j) + ...
    enddo
  enddoall
  doall i = 1, n
   s1: call x(a,i,2,n)
  enddoall
end parallel section
```

Fig. 3. Parallelized program

In this paper, we present an algorithm to discover and analyze interprocedural induction variables in parameters of procedures.

The loops of recursive procedure calls or returns to form interprocedural induction variables are *implicit*, because they do not exist in the abstract syntax trees of the program. Moreover, 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 variable analysis first needs to recover, identify and analyze these implicit loops. We have extended the Full Program Representation (FPR) graph [12] for this purpose. The contributions of this paper are:

- the techniques to identify and analyze the unique loop structure of recursive calls and returns and
- the complete algorithm to identify and analyze interprocedural induction variables.

We use FORTRAN as the base program model in this paper, but allow recursive procedure calls. We 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 loops, 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 equal to zero.

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 et al. [12] for this purpose.

Suppose that 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 ($0 \leq i \leq n-1$) 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. $S = \cup_{i=0}^{n-1}\{s_i\}$ and $R = \cup_{i=0}^{n-1}\{r_i\}$. Let B_i be the set of branch nodes in the control flow graph of procedure i and $B = \cup_{i=0}^{n-1} B_i$. 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, respectively. The edge set E of G is defined as follows:

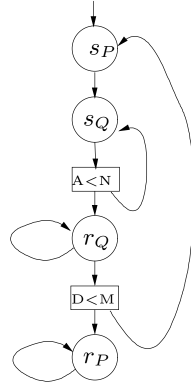
1. If procedure k calls procedure i (k and i are not necessarily different) at call site cs, and there is a control flow path from a node $a \in A_k$ of procedure k to cs which does not contain any other call statements or branch nodes, there is an edge $(c, s_i) \in E$ where $c = a$ if $a \in B_k$, or $c = s_k$ if a is $entry_k$.
2. If procedure k calls procedure i (k and i are not necessarily different) 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 branch nodes, there is an edge $(r_i, c) \in E$ where $c = a$ if $a \in B_k$, or $c = r_k$ if a is $exit_k$.
3. If procedure k calls procedures i and j (i and j are not necessarily different) at call sites cs_1 and cs_2, respectively, and there is a control flow path from cs_1 to cs_2 which does not contain any other call statements or branch nodes, 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 branch nodes, there is an edge $(c_1, c_2) \in E$ where $c_1 = a_1$ if $a_1 \in B_k$, or $c_1 = s_k$ if a_1 is $entry_k$, and $c_2 = a_2$ if $a_2 \in B_k$, or $c_2 = r_k$ if a_2 is $exit_k$.

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

```
subroutine P(C,D,M,E)
  D=D+2
  G = C
  call Q(G,D,M,E)
  if (D<M) then
    call P(C,D,M,E)
  endif
end

subroutine Q(A,B,N,F)
  A=A+1
  F=F+3
  if (A<N) then
    call Q(A,B,N,F)
  endif
end
```

(a) Program



(b) EFPR graph

Fig. 4. Nested recursive calls and returns

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 with each other rather than nested and, thus, the EFPR graph may be irreducible. Moreover, a procedure may be called at more than 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 the induction variable analysis. This can be done by checking first the call graph and then the EFPR graph of the program. The purpose of the checking is to rule out the programs not suitable for interprocedural induction variable analysis. In this section, we first discuss the call graph checking and the EFPR graph checking. We then present the typical loop structure of recursive calls and returns for interprocedural induction variable analysis.

3.1. Call Graph Checking

We assume that the call graph is a connected graph. The call graph of a program is a directed multi-graph (V, E), where the node set V is the set of procedures and E the set of edges such that $(p, q) \in E$ if and only if there is a call site in procedure

p which calls procedure q.

There are four cases in which we will rule out the program for interprocedural induction variables analysis:

1. There no no cycles in the call graph. In this case, there will be no loops to form interprocedural induction variables.
2. There are cycles in the call graph, but at least one of the followings is true:
   (a) There is a node with three or more incoming edges. If a node has three or more incoming edges, it is impossible for its parameters to become induction variables. Since the call graph is connected and all the other procedures may either call (directly or indirectly) this procedure or be called (directly or indirectly) by it. We conservatively assume that all parameters of all procedures are complex variables.
   (b) There is a node which is not the header of a natural loop, but has two or more incoming edges[b]. The parameters of such a procedure cannot be induction variables. For the same reason as above, we conservatively assume that there are no induction variables in the parameters of all procedures.
   (c) After the checking above, the call graph must have natural loops with their headers to be the only nodes with 2 incoming edges and all the other nodes have only one incoming edge. At this step, we need to check whether all the natural loops are either nested or disjoint. If there are two natural loops partially overlapped (none of them completely includes the other and they have common nodes), we conservatively assume that all parameters of all procedures are complex variables.

After the call graph checking above, the compiler needs to build the Extended Full Program Representation (EFPR) graph for further checking.

3.2. EFPR Graph Checking

First of all, we do not consider the programs which have implicit loops and explicit loops overlapped with each other in the EFPR graph. Consider the program in Fig. 5(a) where procedures `P` and `Q` call each other and there is an explicit loop surrounding the call of procedure `Q` in procedure `P`. The EFPR graph of this program is shown in Fig. 5(b). This example shows that it is possible for implicit loops of recursive call to overlap with explicit loops and the EFPR graph is irreducible. To rule out such programs for further analysis, we need to check the branch nodes in a cycle of procedure calls in the EFPR graph. Given a natural loop

$$s_0 \rightarrow ..... \rightarrow s_1 \rightarrow .... \rightarrow \cdots \rightarrow s_{n-1} \rightarrow .... \rightarrow s_0$$

with header s_0, where a "....." represents a path containing zero or more consecutive

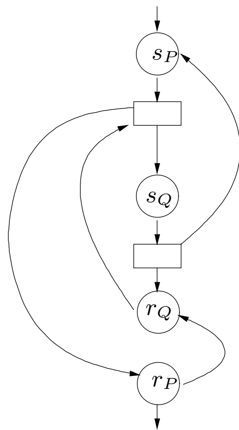
[b] A *header* d is the head of a back edge $n \rightarrow d$ (n is the tail of the edge.). An edge $n \rightarrow d$ is a back edge if d dominates n. d dominates n if all the paths from the entry node of the graph (the main program in the call graph) to n include d. Given a back edge $n \rightarrow d$, the natural loop with header d is the set of nodes which can reach n without going through d. The natural loop is the graph model for the explicit loops like `for` or `DO` loops.

```
program main
   call P(...)
end

subroutine P(...)
    do ...
      call Q(...)
    enddo
end

subroutine Q(...)
    if ...
      call P(...)
    endif
end
```

(a) Program



(b) EFPR graph

Fig. 5. Checking nested implicit and explicit loops

branch nodes between the start nodes, we only need to check that none of these branch nodes is the header of a natural loop. This can be done by checking that each of the branch nodes has only one incoming edge.

After this checking, we can assume that a natural loop whose header is a start node of a procedure does not include any branch node with more than one incoming edges in the EFPR graph.

3.3. Typical Loop Structure of Recursive Calls and Returns

As in the intraprocedural induction variable analysis, the analysis for interprocedural induction variables starts with the innermost loop. After the call graph checking and the EFPR graph checking above, the structure of a typical innermost loop of recursive calls is illustrated in Fig. 6, 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 edge in Fig. 6 represents a *separate path* from the source to the destination in the EFPR graph. Note that it is possible that there are multiple different paths from b_i to $r_{(i-1) \bmod n}$ due to the branch nodes between them. For the same reason, there may be multiple paths from r_i to $r_{(i-1) \bmod n}$. Note the cycles from r_{n-1} to r_0 and back to r_{n-1} formed by the dotted edges in Fig. 6. Each of these cycles is a control flow of the procedure returns and is called a *dual loop* of the loop of recursive calls.

The *trip count* of the loop of recursive calls, denoted t_c, is the number of times the control goes through the loop header s_0. Since s_0 is visited at least once, we have $t_c \geq 1$. There will be $t_c - 1$ full trips of the cycle from s_0 to s_{n-1} and back to s_0. The last trip is a partial trip and one of the branch nodes $b_1, \cdots, b_{n-1}, b_0$ will take the branch off the cycle. The first branch node that takes the branch off the cycle is called the *break point* of the loop. If the break point is branch node b_j, the

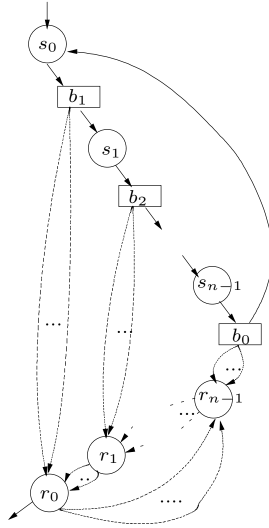


Fig. 6. Structure of typical innermost loop of recursive calls and its dual loops

control flow will take one of the paths to reach $r_{(j-1) \bmod n}$. Then it will make $t_c - 1$ full trips of the cycles from $r_{(j-1) \bmod n}$ back to $r_{(j-1) \bmod n}$ followed by a partial trip from $r_{(j-1) \bmod n}$ to r_0 at which the entire loop exits.

Obviously, the innermost loop of recursive calls should not contain any loop of other recursive calls. In Fig. 6, this means that none of $s_1, \cdots, s_{n-1}$ is the header of a natural loop. (Note that the branch nodes $b_0, \cdots, b_{n-1}$ cannot be headers of natural loops due to the EFPR graph checking above.)

For a loop of recursive calls to be the innermost one, any of its dual loop should not contain any loop of recursive call. That is, any cycle from r_0 to r_{n-1} and back to r_0 in the EFPR graph should not contain any start node with more than one incoming edges.

The third requirement is that any path from the branch node in the loop of recursive calls to the exit of the procedure to which it belongs does not contain any loop of recursive calls. That is, any path from b_i to $r_{(i-1) \bmod n}$ for any $0 \leq i \leq n-1$ in the EFPR graph should not contain any start node with more than one incoming edges. Therefore, the innermost loop of recursive call is defined 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 edges and (3) for any $0 \leq i \leq n-1$, 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 edges. 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 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 $s_0 \to \cdots \to s_1 \cdots s_{n-1} \to \cdots \to s_0$ as defined in Definition 1 and illustrated in Fig. 6.
2. Construct the interprocedural factored use-def (IFUD) graph of the procedures $0, 1, \cdots, n-1$. Apply the modified Tarjan's algorithm [5, 6] 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 a basic induction variable, and the trip count and the break point obtained.
4. Check the EFPR graph to see if there is only one path from r_j to $r_{(j-1) \bmod n}$ for every $j = 0, \cdots, n-1$. If that is the case for all $j = 0, \cdots, n-1$ (i.e. there is only one single dual loop), continue to find induction variables in output parameters caused by the dual loop as follows; otherwise, go to 5:
   (a) Check if there is only one path in the EFPR graph from the break point to the return node of the procedure. If there are multiple paths, go to 5.
   (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.
   (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 induction variable and the trip count as the loop of recursive calls.
5. Coalesce this innermost loop of recursive calls and its dual loops in the EFPR graph.

We next describe each step of the algorithm in detail. We also use the program in Fig. 7 as the working example to illustrate the algorithm.

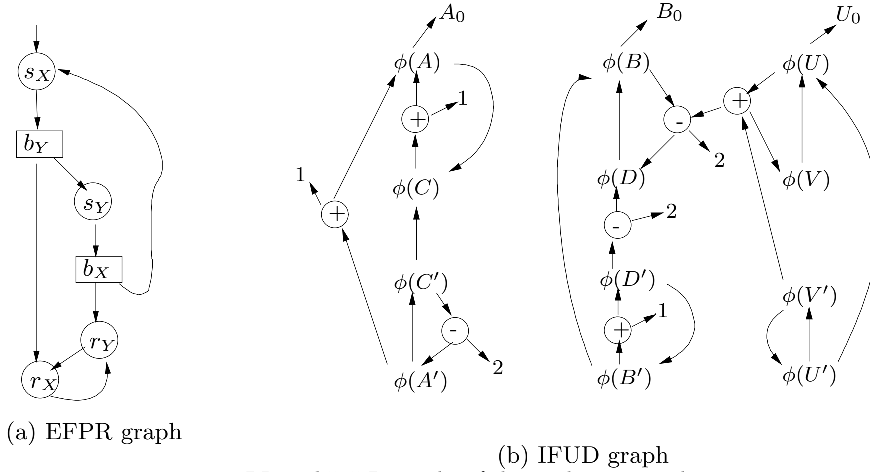
```
program main
   call X(1,9,1)
end

subroutine X(A,B,U)
    A=A+1
c1: if (A<B) then
     call Y(A,B,U)
     B=B+1
    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
```

Fig. 7. Program of working example

The EFPR graph of this program is shown in Fig. 8(a).



(a) EFPR graph

(b) IFUD graph

Fig. 8. EFPR and IFUD graphs of the working example

4.1. Finding the innermost loops of recursive calls (Step 1)

The innermost loop of recursive calls is defined in Definition 1. The algorithm of finding natural loops described in [13] 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 its dual loops as illustrated in Fig. 6.

4.2. Detecting Interprocedural Induction Variables (Step 2)

Although the EFPR graph enables the compiler to find the innermost loop of recursive calls and its dual loops, it does not contain data flow information among the parameters of the procedures. To detect the interprocedural induction variables among these parameters, the compiler still needs an interprocedural factored use-def (IFUD) graph. The basic technique of induction variable detection is the same as the *intra*procedural induction variable detection [5, 6] and is summarized as follows:

- Form a factored use-def (FUD) graph for the variables using the static single-assignment (SSA) representation of the program where each variable has a single definition [14]. SSA representation uses ϕ functions at join points. An edge of the FUD graph is from a use of a variable to its unique definition.
- Use the modified Tarjan's algorithm to traverse the FUD graph to find a loop which (1) has a loop header ϕ-term with only two sources: one from the initial value and the other the back edge of the loop and (2) all the operations in the loop are either fetches, stores of scalar variables or additions of constant or loop invariant values. Such a loop defines a basic induction variable and the induction step is the summation of the values added within the loop. The modified Tarjan's algorithm makes the traverse more efficient because

the types and the values of sub-expressions will be available when they are needed.

The details of the technique can be found in [6].

We next define the interprocedural factored use-def (IFUD) graph.

First of all, we see each parameter of a procedure as both input and output variables because call-by-reference is used in our model. At the entry node of the procedure, the value of the parameter is one of the values passed at the call sites in the program. Therefore, the parameter at the entry of the procedure is modeled by a ϕ-term with the passed values as its sources. On the other hand, the value of the parameter at the exit node of the procedure is one of the values reaching from within the procedure. Hence, the parameter at the exit node is also modeled by a ϕ-term. This ϕ-term defines the value to be passed to whatever the actual parameter bound at a call site.

For this reason, for each parameter X of each procedure P, there are an *input ϕ-term*, denoted $\phi(X)$, and an *output ϕ-term*, denoted $\phi(X')$, in the IFUD graph.[c] The IFUD graph is obtained by merging the factored use-def graphs of the procedures through these input and output ϕ-terms of the formal parameters. The IFUD graph of the working example is shown in Fig. 8(b).

Once the IFUD graph is established, the compiler can start the modified Tarjan's algorithm from any input parameter of the procedures. For example, starting from $\phi(U)$ in Fig. 8(b), the compiler will first find that the loop $(\phi(B), -, \phi(D), \phi(B))$ defines a *basic* induction variable with induction step equal to -2. Then it will conclude that the $(\phi(U), +, \phi(V), \phi(U))$ does not form an induction variable because the sum of values added in the loop is not constant. Applying the modified Tarjan's algorithm from $\phi(A)$ will find another basic induction variable formed by the loop $(\phi(A), \phi(C), +, \phi(A))$ with induction step equal to 1. As a result, the compiler will mark input parameters A and C, B and D, as induction variables and input parameters U and V, as complex variable.

4.3. Trip Count and Break Point (Step 3)

In this step, the compiler finds the trip count t_c and the break point of the loop of recursive calls. The purpose is to find out the number of times each procedure is called so that the closed-form representations of the induction variables in input parameters obtained in the previous step can be formulated. The break point also affects the analysis of induction variables in the output parameters in the subsequent steps.

To simplify presentation, we assume that there is at most one branch node between successive start nodes in the loop of recursive calls defined in Definition 1.

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] To simplify presentation, we assume the names of formal parameters of different procedures are different and we do not need to prefix procedure names to distinguish them.

C_i is a constant logic TRUE. Our purpose is to find out the number of times each of s_i $(i = 0, \cdots, n-1)$ is visited.

Trip Count

We have defined the trip count, t_c, as the number of times s_0 is visited.

Now 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. Note that these input parameters have already been marked as loop invariant, 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. The input parameters are then replaced by the linear forms of the basic 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 induction variable k is a non-negative integer starting from 0 when s_0 is first visited and is incremented each time s_0 is re-visited. 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 a single variable and it may or may not have solutions. 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 $k_{max} \geq 0$ 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 sequence of values $0, 1, \cdots, k_{max}, k_{max}+1$ when s_0 is visited. Hence, we have

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

We define $k_{\max} = -1$ if the integer linear system does not have solution so that we can have a uniform equation for t_c: $t_c = k_{\max} + 2$.

Consider the working example in Fig. 7. Its EFPR graph in Fig. 7(a) shows that the innermost loop $s_X \rightarrow b_Y \rightarrow s_Y \rightarrow b_X \rightarrow s_X$ contains two branch nodes, b_Y and b_X. After analyzing the IFUD graph in Fig. 7(b), the compiler can determine that both $\phi(A)$ and $\phi(B)$ are induction variables and their values as linear functions of the basic induction variable k are $A(k) = A_0 + k$ and $B(k) = B_0 - 2k$, respectively. So are the $\phi(C)$ and $\phi(D)$ whose values are expressed by $C(k) = A(k)+1 = A_0+k+1$ and $D(k) = B(k) = B_0 - 2k$, respectively. The condition of node b_Y in terms of the values of $\phi(A)$ and $\phi(B)$ is $A(k)+1 < B(k)$, taking the data flow from the entry of procedure X to branch node b_Y (i.e. `if (A<B)`) into consideration. Similarly, the condition of node b_X is $C(k) < D(k) - 2$. Therefore, the conditions of b_Y and b_X in terms of k are $C_Y(k) \equiv 3k+1 < B_0 - A_0$ and $C_X(k) \equiv 3k+3 < B_0 - A_0$, respectively. The largest k satisfying $C_Y(k) \wedge C_X(k) \wedge (k \geq 0)$ is $k_{max} = \lfloor \frac{B_0 - A_0 - 3}{3} \rfloor = \lfloor \frac{9-1-3}{3} \rfloor = 1$. The trip count is $t_c = 3$.

Break Point

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

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

Once the trip count and the break point are found, the closed-form representation of the induction variables in the input parameters can be derived.

Suppose that the break point is b_j ($0 \le j \le n-1$) in Fig. 6. Then, each of the start nodes $s_0, \cdots, s_{(j-1) \bmod n}$ will be visited $t_c = k_{\max} + 2$ times with the basic induction variable k taking the sequence of values $0, 1, \cdots, k_{max}+1$, while each of the remaining start nodes, $s_j, \cdots, s_{n-1}$ ($j \neq 0$), will be visited $t_c - 1 = k_{\max} + 1$ times with k taking the sequence of values $0, 1, \cdots, k_{max}$. The break point b_j actually divides the procedures into two groups:

- Group I: procedures $0, \cdots, (j-1) \bmod n$, each of which is called $t_c = k_{\max}+2$ times, and
- Group II: procedures $j, \cdots, n-1$ ($j \neq 0$), each of which is called $t_c - 1 = k_{\max} + 1$ times.

If $j = 0$, Group II is empty and all procedures belong to Group I.

Given an induction variable I found in the input parameters, its closed-form representation is

$$I_{in} = \{I(k) = I_0 + k \cdot s \mid 0 \le k \le k_{\max} + 1\} \tag{1}$$

or

$$I_{in} = \{I(k) = I_0 + k \cdot s \mid 0 \le k \le k_{\max}\} \tag{2}$$

if I belongs to a procedure in Group I or Group II, respectively. Here, I_0 and s are the initial value and the induction step, respectively, obtained in the previous step and $I(k)$ denotes the linear function for the values of the induction variable I.

In our working example, the break point is b_X (i.e. b_0) and both procedures belong to group I. Putting it all together, the closed-form representations of four induction variables in the input parameters are:

$$\begin{aligned}
A_{in} &= \{A(k) = A_0 + k \mid 0 \le k \le k_{max} + 1\} \\
B_{in} &= \{B(k) = B_0 - 2k \mid 0 \le k \le k_{max} + 1\} \\
C_{in} &= \{C(k) = A_0 + k + 1 \mid 0 \le k \le k_{max} + 1\} \\
D_{in} &= \{D(k) = B_0 - 2k \mid 0 \le k \le k_{max} + 1\}
\end{aligned}$$

with $k_{\max} = 1$.

4.4. Induction Variables of Dual Loop (Step 4)

After the analysis of induction variables in the input parameters, the compiler tries to find possible induction variables in the output parameters formed by its dual loop.

Given an innermost loop of recursive calls defined in Definition 1, there can be several paths from r_i to $r_{(i-1) \bmod n}$, $(0 \leq i \leq n-1)$, in the EFPR graph. If there are multiple paths from r_i to $r_{(i-1) \bmod n}$, the factored use-def chains from the output parameters of $r_{(i-1) \bmod n}$ to that of r_i will go through ϕ-terms. As a consequence, none of the output parameters of all the procedures can be an induction variable. The entire Step 4 should exit and all the output parameters should be marked as complex variables. In other words, the compiler will continue to detect induction variables in the output parameters in Steps 4(a), 4(b) and 4(c) only if there is only one path r_i to $r_{(i-1) \bmod n}$ for every $i = 0, \cdots, n-1$. These steps are described as follows:

4.4.1. Checking Paths from Break Point to Return Node (Step 4(a))

In the discussion from here, we suppose that the break point of the loop of recursive calls shown in Fig. 6 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.

4.4.2. Determining Initial Values of Output Parameters (Step 4(b))

At this step, the algorithm uses the IFUD graph to find the expressions for the initial values of the output parameters of the procedure $(j-1) \bmod n$. Since there is a single path from the break point b_j to the return node $r_{(j-1) \bmod n}$ in the EFPR graph, a single expression in terms of the input parameters of the procedure $(j-1) \bmod n$ can be found. If the expression of an output parameters contains a complex input parameter, the output parameter does not have a constant initial value and should be marked as a complex variable.

Continuing our working example with the break point b_X, the final values of input parameters of C and D are $C(k_{max}+1)$ and $D(k_{max}+1)$, respectively. Following the IFUD chains corresponding to the path (s_Y, b_X, r_Y) in the EFPR graph, the compiler can find the unique initial values of output parameters C' and D', denoted C'_0 and D'_0, as follows: $C'_0 = C(k_{max}+1) = A_0 + k_{\max} + 2$ and $D'_0 = D(k_{max}+1) - 2 = B_0 - 2k_{\max} - 4$.

4.4.3. Detecting Induction Variables in Output Parameters (Step 4(c))

This step is similar to Step 2 of the algorithm. The compiler applies the Tarjan's algorithm to the IFUD chains of the output parameters corresponding to the dual loop to find possible induction variables. In our working example, the result is that both C' and D' are induction variables with induction steps -2 and 1, and are

expressed as $C'(k') = C'_0 - 2k'$ and $D'(k') = D'_0 + k'$, respectively. Here, k' is the basic induction variable of the *dual loop*.

Similar to the basic induction variable of loop of recursive calls k, the basic induction variable of the dual loop k' starts with zero when $r_{(j-1) \bmod n}$ is first visited. It is then incremented every time $r_{(j-1) \bmod n}$ is re-visited.

Note that for each procedure call there must be a return of the procedure. The number of the returns of a procedure must be equal to the number of the calls of the same procedure. Therefore, each of the return nodes $r_{(j-1) \bmod n}, \cdots, r_0$ will be visited $t_c = k_{\max} + 2$ times with the basic induction variable of the dual loop k' taking the sequence of values $0, 1, \cdots, k_{\max} + 1$, while each of the remaining return nodes $r_{n-1}, \cdots, r_j$ $(j \neq 0)$ will be visited $t_c - 1 = k_{\max} + 1$ with k' taking the sequence of values $0, 1, \cdots, k_{\max}$.

Given an induction variable I' found in the output parameters, its closed-form representation is

$$I'_{out} = \{I'(k') = I'_0 + k' \cdot s' \mid 0 \leq k' \leq k_{\max} + 1\} \tag{3}$$

or

$$I'_{out} = \{I'(k') = I'_0 + k' \cdot s' \mid 0 \leq k' \leq k_{\max}\} \tag{4}$$

if I' belongs to a procedure in Group I or Group II, respectively. Here, I'_0 and s' are the initial value and the induction step, respectively, and $I'(k')$ denotes the linear function for the values of the induction variable I'.

Each procedure return corresponds to a particular call. If a parameter I is found to be induction variables as both input and output parameters, we are also interested in the input and output values of I of the *corresponding* call and return. Therefore, we need to find a closed form for I' using the basic induction variable of the loop of recursive call k, instead of the basic induction variable of the dual loop k'.

Given a procedure i called recursively, its first return corresponds to its last call. In our framework, this means that the first visit of r_i corresponds to the last visit of s_i.

From the definitions of k and k' and the discussion above, we can reach the following theorem:

Theorem 1 *If procedure i belongs to Group I, the visit of r_i with $k' = l$ $(0 \leq l \leq k_{\max} + 1)$ represents the return corresponding to the call represented by the visit of s_i with $k = k_{\max} + 1 - l$.*

Proof. Since procedure i belongs to Group I, s_i is visited $k_{\max} + 1$ times with k taking the sequence of values $0, 1, \cdots, k_{max} + 1$. r_i is also visited $k_{\max} + 1$ times with k' taking the sequence of values $0, 1, \cdots, k_{max} + 1$. Since the first return of procedure i corresponds to its last call, the visit of r_i with $k' = 0$ represents the return corresponding to the call represented by the visit of s_i with $k = k_{max} + 1$. This proves the theorem. □

The similar theorem for the procedures in Group II is as follows:

Theorem 2 *If procedure i belongs to Group II, the visit of r_i with $k' = l$ ($0 \leq l \leq k_{\max}$) represents the return corresponding to the call represented by the visit of s_i with $k = k_{\max} - l$.*

Given an induction variable I' in the output parameters of procedure i, we use $I''(k)$ to denote the function for its value at the return corresponding to the call represented by the visit of s_i with k. According to Theorem 1 or Theorem 2, that return must be represented by the visit of r_i with $k' = k_{\max}+1-k$ or $k' = k_{\max}-k$, depending on whether procedure i belongs to Group I or Group II, respectively. In other words, we have

$$I''(k) = I'(k_{\max} + 1 - k) \tag{5}$$

or

$$I''(k) = I'(k_{\max} - k) \tag{6}$$

if procedure i belongs to Group I or Group II, respectively. Here, function $I'(k')$ is defined in Eq.(3) and Eq.(4).

Now we can have the closed-form representation of the output induction variable I' using the same basic induction variable k as the input induction variable I as follows:

$$I''_{out} = \{I''(k) = I'_0 + (k_{\max} + 1 - k) \cdot s' \mid 0 \leq k \leq k_{\max} + 1\} \tag{7}$$

or

$$I''_{out} = \{I''(k) = I'_0 + (k_{\max} - k) \cdot s' \mid 0 \leq k \leq k_{\max}\} \tag{8}$$

if I' belongs to a procedure in Group I or Group II, respectively.

In our working example, output parameters A' and B' are also induction variables. Their initial values, A'_0 and B'_0, can be obtained by following the IFUD chains corresponding to the path (r_Y, r_X) in the duel loop and 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 $B'(k') = B'_0 + k'$. By using Eq.(5) (since all procedures belong to Group I), we reach the following representations:

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

where $k_{max} = 1$.

4.5. Loop Coalescing (Step 5)

The last step (Step 5) in the **while** loop of the algorithm is to coalesce the innermost loop of recursive calls and its dual loop into a simple procedure which summarizes the effect of the recursive procedure call.

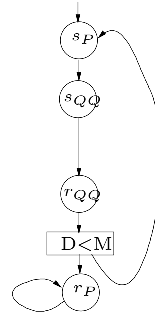
Given the the innermost loop of recursive calls, $s_0 \to \cdots \to s_1 \cdots s_{n-1} \to \cdots \to s_0$, the compiler creates a new procedure to replace all the procedures involved,

namely, procedures $0, 1, \cdots, n-1$. (Recall that s_i is the start node of procedure i.) The compiler can create the simple procedure as follows:

1. The simple procedure has the same formal parameters as procedure 0.
2. For each parameter A of procedure 0
   (a) if both input and output parameters of A^d are induction variables or loop invariants expressed as $A(k)$ and $A''(k)$, respectively, and at least one of them is an induction variable,
      (i) calculate $AS = A''(0) - A(0)$. This gives the *increment* of A as the side effect of the recursive call.
      (ii) create a statement `A = A + AS` in the simple procedure
   (b) if both input and output parameters of A are loop invariants, do nothing;
   (c) otherwise, mark both input and output parameters of A as a complex variable.

```
subroutine QQ(A,B,N,F)
  A=A+(N-A)
  F=F+3*(N-A)
end
```

(a) Simple procedure QQ



(b) EFPR graph after loop coalescing

Fig. 9. Loop coalescing

Let us go back to the example of nested loops of recursive calls in Fig. 4(a) again. The induction variables analysis for the innermost loop reveals that A and F are induction variables, and B and N are loop invariant variables. All of output parameters A', F', B' and N' are loop invariant variables. We have

$$\begin{aligned} A_{in} &= \{A(k) = A_0 + k \mid 0 \le k \le k_{max} + 1\} \\ F_{in} &= \{F(k) = F_0 + 3k \mid 0 \le k \le k_{max} + 1\} \end{aligned}$$

and $k_{max} = N_0 - A_0 - 2$. We also have $A''(0) = A_0 + k_{max} + 2$ and $F''(0) = F_0 + 3k_{max} + 6$. Therefore, increments of A and F are $A_s = N_0 - A_0$ and $F_s = 3(N_0 - A_0)$. The simple procedure to replace the innermost loop of recursive calls and its dual loop is, thus, shown in Fig. 9(a) and the new EFPR graph after this loop coalescing is shown in Fig. 9(b). The new IFUD graph after coalescing is shown in Fig. 10.

The further analysis of the innermost loop with loop header node s_P will reveal that input parameter C and M are loop invariant variables and input variable D is

[d] Recall that each parameter has the input and output ϕ-terms in the IFUD graph representing the input and output parameters of the same name.

Fig. 10. IFUD after loop coalescing

an induction variable with induction step of 2. Input parameter E can also be found to be an induction variable with induction step of $3(M_0 - C_0)$, because the compiler will find that $3(N - A)$ is a loop invariant value when the Tarjan's algorithm searches from $\phi(E)$ in the IFUD graph in Fig. 10.

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 the techniques and the algorithm to discover interprocedural induction variables in procedure parameters formed by loops of recursive calls or returns. We extended the full program representation graph [12] and the factored use-def graph [6] to be used as the framework for the analysis in this paper.

References

1. Z. Li and P.-C. Yew, "Interprocedural analysis for parallel computing," *Proc. 1988 International Conference on Parallel Processing, Vol. II*, Aug. 1988, pp. 225–244.
2. P. Tang, "Exact side effects for interprocedural dependence analysis," *Proc. 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, Jul. 1993, pp. 137–146.
3. Y. Paek, J. Hoeflinger, and D. Padua, "Simplification of array access patterns for compiler optimizations," *Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1998, pp. 60–71.
4. J. Hoeflinger and Y. Paek, "The access region test," *Proc 1999 International Workshop on Languages and Compilers for Parallel Computing*, La Jolla, CA, Aug. 1999,

pp. 271-285.

5. M. Wolfe, “Beyond induction variables,” *Proc. 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1992, pp. 162–174.
6. M. Wolfe, *High Performance Compilers for Parallel Computing* (Addison-Wesley, 1995).
7. F. E. Allen, J. Cocke, and K. Kennedy, “Reduction of operator strength,” in *Program Flow Analysis: Theory and Applications*, eds. S. S. Muchnick and N. D. Jones (Prentice-Hall, 1981) pp. 79–101.
8. J. Cocke and K. Kennedy, “An algorithm for reduction of operator strength,” *Communications of the ACM* **20(11)** (1977) 850–856.
9. Z. Ammarguellat and W. L. Harrison III, “Automatic recognition of induction variables and recurrence relations by abstract interpretation,” *Proc. 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1990, pp. 283–295.
10. M. R. Haghighat and C. D. Polychronopoulos, “Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs,” *Proc. 1993 International Workshop on Languages and Compilers for Parallel Computing*, Aug. 1993, pp. 567–585.
11. M. R. Haghighat, *Symbolic Analysis for Parallelizing Compilers* (Kluwer Academic Publishers, 1995).
12. G. Agrawal, J. Salts, and R. Das, “Interprocedural partial redundancy elimination and its applications to distributed memory compilation,” *Proc. 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1995, pp. 258–269.
13. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools* (Addison-Wesley Publishing Company, 1986).
14. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems* **13(4)** (1991) 451–490.