Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 02-010

Interprocedural Induction Variable Analysis

Peiyi Tang and Pen-chung Yew

February 28, 2002

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

February 19, 2002

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.

Key Words: 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 allow the data dependence analysis to discover loop parallelism for parallel execution.

[*]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.

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 Figure 1(b), formal parameter `k` is an induction variable $\{k_0+2i \mid 0 \leq i \leq \lfloor \mathtt{n}/2 \rfloor - 1\}$ 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: $b(i,k_o), b(i,k_0+2), \cdots$. 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 of accessed in statement `s0` by loop `j` are the odd elements each 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 [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.

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. 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. 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 call and returns and
- the complete algorithm to identify and analyze interprocedural induction variables.

To form interprocedural induction variables, the loop of recursive calls has to conform to certain format. Before starting the algorithm of analysis, the compiler can rule out many programs which do not have interprocedural induction variables by checking their call graphs and the extended FPR graph first.

The rest of the paper is organized as follows. Section 2 describes the program model of the programs to be analyzed and other preliminary backgrounds. Section 3 describes the call graph checking. Section 4 describes our extended FPR graph. Section 5 describes the checking of the extended FPR graph. Section 6 presents the algorithm to find interprocedural induction variables. Section 7 concludes the paper with a discussion of related work.

2 Program Model and Preliminaries

2.1 Program Model

The model of the programs to be analyzed is as follows:

- Parameters of procedures can be array variables or scalar variables. We concentrate on scalar parameters for induction variable analysis.
- Scalar parameters are passed by reference.
- Scalar parameters are variables of integer type.
- To simplify the presentation, we do not consider
  - procedure-valued variables.
  - global variables
  - pointer variables
- The control structures in procedures include `if-then`, `if-then-else` and explicit loops like `for` or `DO`.
- Each procedure has only one return statement which is the end of the procedure.

2.2 Purpose of Analysis

The purpose of the interprocedural induction variable analysis is to classify all the scalar parameters of the procedures to the following three categories:

- loop invariant variables with respect to the corresponding loop of procedure call or return, or
- induction variables with respect to the corresponding loop of procedure call or return, or
- 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.

3 Call Graph Checking

According to the assumption of the program model above, the call graph is static. We can also assume that the call graph is a connected graph because every procedure is assumed to be called at least at one call site. 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 abort the 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 the parameters of all the 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. [1] 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 and the analysis stops here.
   (c) After passing these checks, 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 of checking, we need to check whether all the natural loops are either nested or disjoint. If

[1]A *header* d is the head of a back edge $n \to d$ (n is the tail of the edge.). An edge $n \to d$ 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 $n \to d$, 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.

there are two natural loops partially overlapped (none of them completely includes the other and they have common nodes), the analysis stops here, because the implicit loops of recursive calls do not conform to the structure of explicit `for` and `DO` loops.

After passing these call graph checkings, the compiler needs to build an Extended Full program Representation (EFPR) graph to further rule out the programs which cannot have interprocedural induction variables.

4 Extended Full Program Representation (EFPR) Graph

The Full Program Representation Graph was proposed by Agrawal el al [12] for interprocedural partial redundancy elimination. We extended it to include the branch nodes and to capture the full control flow of the whole program.

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:

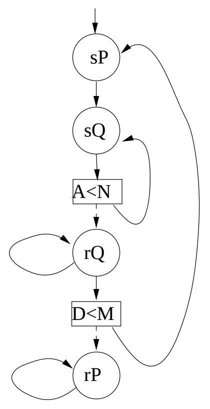
1. If procedure k calls procedure i at call site cs, and there is a control flow path from a node $a \in A_k$ of procedure k to cs which does not contain any other call statements or branch nodes, there is an edge $(c, s_i) \in E$ where $c = a$ if $a \in B_k$ or $c = s_k$ a is $entry_k$.
2. If procedure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

Figure 2(b) shows the EFPR graph for the program in Figure 2(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

Figure 2: Nested Recursive Calls and Returns

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

5 Checking of EFPR Graph

First of all, we do not consider the programs which have implicit loops and explicit loops overlapped with each other. Consider the program in Figure 3(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 Figure 3(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

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

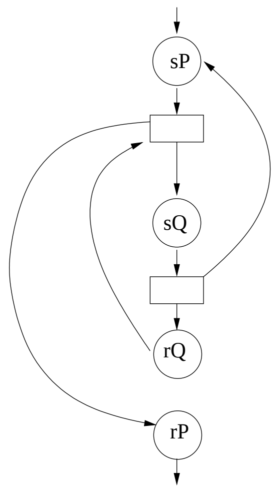
with header s_0, where a ".....” represents a path containing zero or more consecutive branch nodes between the start nodes, we only need to check that none of these branch nodes is the header of a natural loop. This can be done by checking that each of the branch nodes has only one incoming edge.

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 edge in the EFPR graph.

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

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

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



(b) EFPR Graph

(a) Program

Figure 3: Checking nested implicit and explicit loops

6 Algorithm for Interprocedural Induction Variable Analysis

As in the *intra*procedural induction variable analysis, the analysis for *inter*procedural induction variables starts with the innermost loop.

The structure of an innermost loop of recursive calls is illustrated in Figure 4, 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, $0 \le i \le n-1$. A dotted edge in Figure 4 represents a *different 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} up to r_0 and back to r_{n-1} formed by the dotted edges in Figure 4. Each of these cycles is a control flow of procedure returns and is called a *dual loop* of the loop of recursive calls.

The *trip count* of the loop of recursive calls, denoted as 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 $t_c \ge 1$. There will be $t_c - 1$ full trips of the cycle from s_0 up 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 control flow will take one of the paths to reach $r_{j-1 \bmod n}$. Then it will make a partial trip from $r_{j-1 \bmod n}$ to r_0 and then make $t_c - 1$ full trips of the cycle from

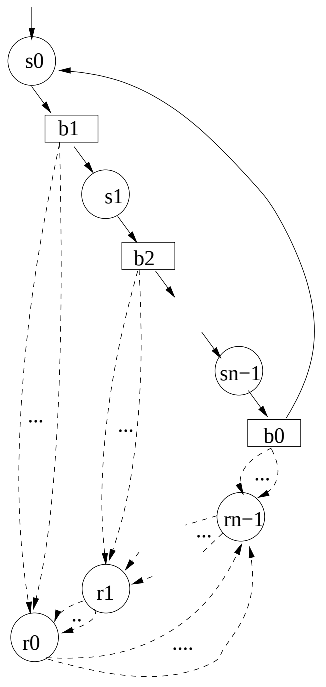


Figure 4: Structure of a typical innermost loop of recursive calls and its dual loop

r_0 back r_0.

Obviously, the innermost loop of recursive calls should not contain any loop of another recursive calls. In Figure 4, 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 either after the checking described in Section 5.)

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 return node of the procedure to which it belongs does not contain any loop of recursive calls. That is, any path from b_i to $r_{i-1 \bmod n}$ in the EFPR graph should not contain any start node with more than one incoming edge. 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, $s_0 \to \cdots \to s_1 \cdots s_{n-1} \to \cdots \to s_0$, 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.*

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 4.
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 obtained.
4. Check the EFPR graph to see if (1) 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. If there are multiple paths, go to Step 5; otherwise continue with Step 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.
   (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 5 as the working example to illustrate the algorithm.

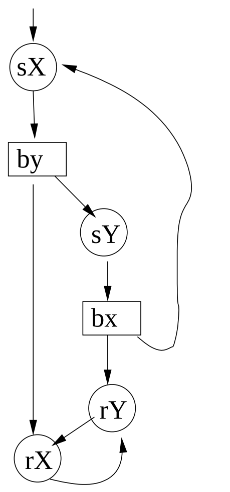
The EFPR graph of this program is shown in Figure 6(a)

6.1 Finding the innermost loops of recursive calls

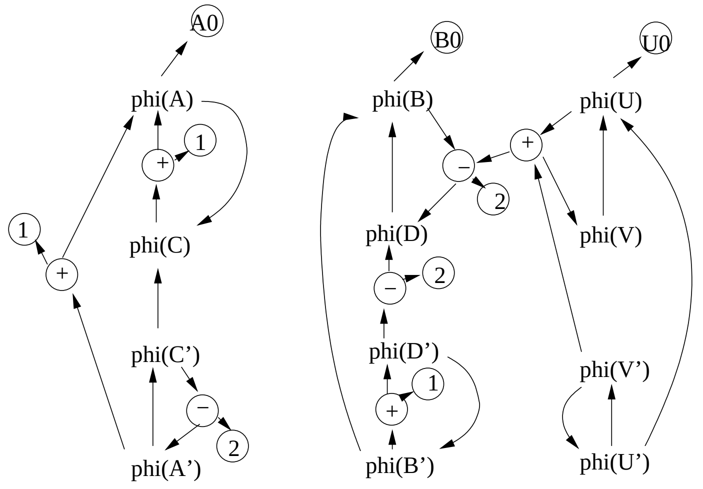
The innermost loop of recursive calls is defined in Definition 1. The algorithm of finding natural loops described in [13] can be used to find all the natural loops in the EFPR graph. Using Definition 1, the compiler is able to find the innermost loop of recursive calls and all its dual loops as illustrated in Figure 4.

```
program main                        subroutine Y(C,D,V)
   call X(1,9,1)                        D=D-2
end                                     V=V+D
                                    c2: if (C<D) then
subroutine X(A,B,U)                       call X(C,D,V)
    A=A+1                                 C=C-2
c1: if (A<B) then                       endif
     call Y(A,B,U)                  end
    endif
end
```

Figure 5: Program of Working Example



(a) EFPR Graph



(b) IFUD Graph

Figure 6: EFPR and IFUD graphs of the working example

6.2 Detecting Interprocedural Induction Variables

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 for the *intra*procedural induction variables 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 loop induction variable and the induction step is the summation of the values added within the loop. The modified Tarjan's algorithm makes the traverse efficient because the type and the value of a sub-expression will be available when it is 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.[2] 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 Figure 6(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 Figure 6(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

[2]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.

the sum of values added in the loop is not constant. Applying the modified Tarjan's algorithm from $\phi(A)$ will find that 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.

6.3 Trip Count and Break Point

The purpose of this step of the algorithm is to find the trip count t_c and the break point of the loop of recursive calls.

To simplify presentation, we assume that there is at most one branch node between successive start nodes of procedures 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_i is a constant logic TRUE.

Trip Count

We have defined the trip count, t_c, as the number of times the control visits s_0.

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 variable in Step (c) 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 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 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 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 $0 \leq k \leq t_c - 1$.

Consider the working example in Figure 5. Its EFPR graph in Figure 5(a) shows that the innermost loop $s_X \to b_Y \to s_Y \to b_X \to s_X$ contains two branch node, b_X and b_Y. After analyzing the IFUD graph in Figure 5(b), the compiler can determine that both $\phi(A)$ and $\phi(B)$ are induction variables and their values in terms of the basic loop 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 $D(k) = B(k) = B_0 - 2k$. The condition of node b_X 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_X in X into consideration. Similarly, the condition of node b_Y is $C(k) < D(k) - 2$. Therefore, the conditions of b_X and b_Y in terms of k are $C_X(k) \equiv 3k + 1 < B_0 - A_0$ and $C_Y(k) \equiv 3k + 3 < B_0 - A_0$, respectively. The largest k satisfying $C_X(k) \wedge C_Y(k)$ is $k_{max} = \lfloor \frac{B_0 - A_0 - 3}{3} \rfloor = \lfloor \frac{9-1-3}{3} \rfloor = 1$.

Break Point

According to the definition of k_{max} above, the last trip carrying with $k_{max} + 1$ 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 $1 \leq j \leq n - 1$.

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.

Putting it all together, the four induction variables in the input parameters 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 $k_{\mathrm{max}} = 1$.

6.4 Induction Variables of Dual Loop

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 output parameters in Steps 4(a), 4(b) and 4(c) of the algorithm only if there is only one path r_i to $r_{i-1 \bmod n}$ for all $i = 0, \cdots, n-1$. These steps are described as follows:

6.4.1 Checking Paths from Break Point to Return Node

Assume that the break point of the loop of recursive calls shown in Figure 4 is b_j. This step (Step 4(a)) checks 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. As a consequence, all the output parameters of the procedures cannot be induction variables and the algorithm goes to Step 5 immediately.

6.4.2 Determining Initial Values of Output Parameters

At this step (Step 4(b)), the algorithm uses the IFUD graph to find the expressions for the initial values of the output parameters of the procedure. Since there is a single path from the break point to the return node in the EFPR graph, a single expression in terms of the input parameters of the procedure can be found. If the expression of an output parameters contains a complex input parameter, that output parameter cannot be an induction variable because its initial value is not constant.

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: $C'_0 = C(k_{max}+1)$ and $D'_0 = D(k_{max}+1) - 2$.

6.4.3 Detecting Induction Variables in Output Parameters

At this step (Step 4(c)), 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. This step is similar to Step 2. 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 (defined as the number of times r_0 is visited) is the same as the loop of recursive calls. Therefore, the basic induction variable, k', takes values $0, 1, \cdots, k_{max}$ and $k_{max}+1$, one for each trip. The first trip from r_j (assuming $b_{j+1 \bmod n}$ is the break point) to r_0 is a partial trip.

The direction of dual loop is opposite to that of the loop of recursive calls. the relationship between k and k' is $k + k' = k_{max} + 1 = t_c - 1$. These observations are summarized in the following theorem:

Theorem 1 *If the trip count of the innermost loop of recursive calls, $s_0 \rightarrow \cdots \rightarrow s_1 \cdots s_{n-1} \rightarrow \cdots \rightarrow s_0$, is t_c, then the trip count of its dual loop is also 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, 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 $B'(k') = B'_0 + k'$. Note that $k' = (k_{max} + 1) - k$. We then convert the expressions of the output parameters to use k instead of k'.

Putting it all together, the four induction variables in the output parameters in the working example are as follows:

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

where $k_{max} = 1$.

6.5 Nested Induction Variables

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, $s_0 \rightarrow \cdots \rightarrow s_1 \cdots s_{n-1} \rightarrow \cdots \rightarrow s_0$, the compiler creates a new procecure to replace all the procedures involved, namely, procedures $0, 1, \cdots, n-1$ (Recall recall that s_i is the start node of procedure i, $i = 0, 1, \cdots, n-1$.) The compiler can create the simple procedure as follows:

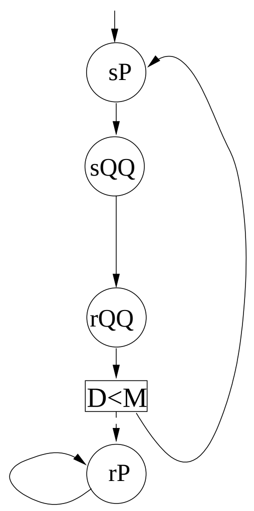
1. The simple procedure has the same formal parameters as procedure 0.
2. For each parameter A of procedure 0
   (a) if both input and output parameters of A[3] are induction variables or loop invariants expressed as $A(k)$ and $A'(k)$, respectively, and at least one of them is an induction variable.
      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.

Let us go back to the example of nested loops of recursive call in Figure 2(a) again. The induction variables analysis for the innermost loop reveals that A and

[3]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.

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

(a) Simple procedure QQ



(b) EFPR graph after loop coalescing

Figure 7: Program 1 after loop coalescing

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

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

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 Figure 7(a) and the new EFPR graph after this loop coalescing is shown in Figure 7(b). The new IFUD graph after coalescing is shown in Figure 8.

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 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 Figure 8.

7 Related Work

Apart from the work on intraprocedural induction variable analysis mentioned in Section 1, this work is also related to the work described as follows:

- The interprocedural induction variable analysis described in this paper can be regarded as an extension of the analysis of interprocedural constants [15, 16].

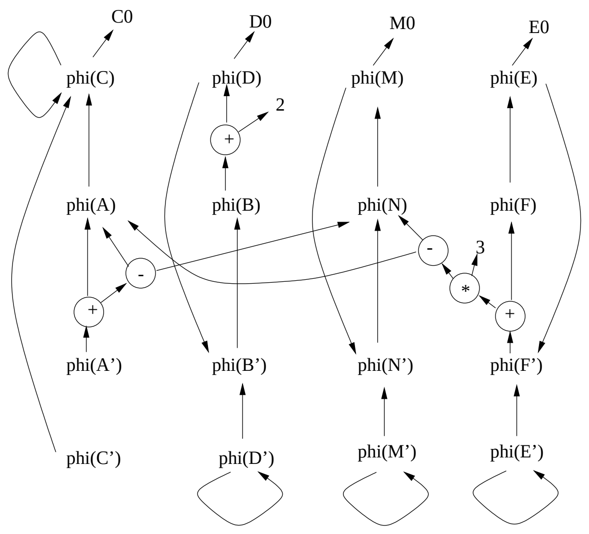


Figure 8: FUD of Program 1 after loop collapsing

The interprocedural induction variable analysis described in this paper goes beyond constant propagation and finds loop invariants and induction variables in formal parameters with respect to the loops of recursive procedure calls and returns.

- This work is also 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.
- This work is related and motivated by the work on array sections or array side effects of procedure calls [1, 17, 2, 3, 4]. To summarize the sections of arrays modified or used by procedures in recursive calls for parallelization, the compiler needs to find the induction variables in both input and output parameters. The work in this paper provides a solution to this problem.

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] Yunheung Paek, Jay Hoeflinger, and David Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 60–71, June 1998.

[4] Jay Hoeflinger and Yunheung Paek. The access region test. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, August 1999.

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

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

[7] F.E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analsis: Threory and Applications*, pages 79–101. Prentice-Hall, 1981.

[8] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, November 1977.

[9] Z. Ammarguellat and W.L. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–295, June 1990.

[10] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 567–585. Lecture Notes in Computer Science, No. 768, August 1993.

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

[12] 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.

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

[14] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[15] D. Callahan, K.D. Cooper, Ken Kennedy, and L. Torczon. Interproceduarl constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices Vol. 21, No. 7*, pages 152–161. ACM, July 1986.

[16] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 90–99, June 1993.

[17] 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.