

# Complete Inlining of Recursive Calls: Beyond Tail-Recursion Elimination

Peiyi Tang

Department of Computer Science  
University of Arkansas at Little Rock  
2801 S. University Ave.  
Little Rock, AR 72204

## ABSTRACT

A compiler optimizing transformation called *complete inlining* to inline and eliminate recursive calls is presented. The complete inlining can eliminate the recursive calls that cannot be eliminated by tail-recursion elimination. It can inline the recursive calls completely which the existing procedure inlining can only inline partially.

## Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*Optimization*; D.3.3 [Programming Languages]: Language Constructs and Features—*Recursion*

## General Terms

Procedure Inlining, Tail-Recursion Elimination

## Keywords

Complete Inlining, Full Control Flow Graph, Call Graph

## 1. INTRODUCTION

Procedure inlining or integration is a program transformation to inline the code of the called procedure to replace the call in the calling procedure [1, 2, 3]. There are basically two benefits of procedure inlining: (1) It eliminates the cost of procedure calls. If the call site is executed frequently, the run-time saving by inlining can be significant. (2) Inlining can bring more opportunities for other compiler optimization and parallelization, because the data flow, control flow and memory use information of the called procedures are fully exposed in the calling procedures [4]. The cost of the procedure inline is the increased code size of the program. There are lots of work focusing on the *profitability* of inlining to avoid code explosion.

The existing inlining algorithms and implementations are effective for inlining non-recursive procedure calls [1, 2, 3, 4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM SE'06 March 10-12, 2006, Melbourne, Florida, USA  
Copyright 2006 ACM 1-59593-315-8/06/0004 ...\$5.00.

If the increased code size is permissible, the current inlining can inline and eliminate all non-recursive calls in a program. For the procedures called recursively, however, the current inlining can inline the recursive calls for a limited number of times, but it cannot eliminate them. Consider the program in Figure 1. The code of  $X$  can be inlined to replace the call of  $X$  in the main program, but it brings another call of  $X$ . Many compilers such as GCC [3] would inline the call of  $X$  several times and then stop inlining. It is in effect unrolling the loop caused by the recursive calls several times. Because the current inlining can inline recursive calls, but cannot eliminate them from the code, we call it *partial inlining*.

Tail-recursion elimination is a compiler optimization to eliminate recursive tail-calls [5, 6]. A tail-call is a call right before the procedure returns. A tail-call is recursive if it is to call the procedure itself. When the recursive call is not the last instruction before the return, it cannot be eliminated by the tail-recursion elimination. The recursive call of  $X$

```
program main
  real a(n,n)
  do i = 1, n
    do j=1, n, 2
      a(i,j) = a(i-1,j)+...
    enddo
    l = 2
  1: call X(a, i, 1, n)
  enddo
end

subroutine X(b, j, k, m)
  real b(m,m)
  if (k+1 < m) then
    k = k + 2
  1: call X(b,j,k,m)
    k = k - 2
  endif
  b(j,k) = ...
end
```

Figure 1: Motivating Example

in procedure  $X$  in Figure 1 cannot be eliminated by tail-recursion elimination, because it is not a tail-call.

In this paper, we present a new compiler optimizing transformation to allow compilers to eliminate the recursive calls that can be eliminated. Figure 2(a) shows the program after our transformation inlines and eliminates all the calls of  $X$  in the program of Figure 1. In particular, the call of  $X$  in the main program is *completely* inlined because the recursive call of  $X$  in the code of  $X$  is eliminated. To distinguish from the current inlining which cannot eliminate recursive calls, we call our transformation *complete inlining*. To the best of our knowledge, there is no prior work that provides such transformation.

After the complete inlining of the program in Figure 1, induction variable analysis [7, 8, 9] can find that variable  $l$  in the inlined program is an induction variable which carries

only even integer values. By using this information, data dependence analysis can find that there are no data dependence cycles between the two statements of updating array  $a$  and the `do` loop  $i$  can be distributed. Furthermore, each of the distributed `do` loops does not carry any data dependence from one iteration to another and, thus, can be parallelized to parallel `doall` loop [10]. Also, since there is no dependence between the two `doall` loops, they can be executed in any order and put into a parallel section. Figure 2(b) shows the parallelized code after all these optimizations.

<pre> program main   real a(n,n)   do i = 1, n     do j = 1, n, 2       a(i,j)=a(i-1,j)+...     enddo   enddo   l = 2 1: vc_X = 0 cl: vc_X = vc_X + 1   if (l+1 &lt; n) then     l = l + 2     goto cl rp: l = l - 2   endif   a(i,l) = ...   vc_X = vc_X - 1   if (vc_X &gt; 0) then     goto rp   endif enddo end </pre> <p>(a) Completely Inlined program</p>	<pre> program main   real a(n,n)   par section   doall i = 1, n     do j = 1, n, 2       a(i,j)=a(i-1,j)+...     enddo   enddoall   l = 2 1: vc_X = 0 cl: vc_X = vc_X + 1   if (l+1 &lt; n) then     l = l + 2     goto cl rp: l = l - 2   endif   a(i,l) = ...   vc_X = vc_X - 1   if (vc_X &gt; 0) then     goto rp   endif enddoall par section end </pre> <p>(b) Parallelized Program</p>
--	---

**Figure 2: Completely Inlined and Parallelized Code**

The organization for the rest of the paper is as follows. Section 2 describes the full control flow graph that is necessary to understand the analysis in this paper. Section 3 describes what kinds of recursive calls can be eliminated and how to find them from the call graph. Section 4 presents the algorithm of complete inlining to inline and eliminate recursive calls. Section 5 discusses the related work and concludes the paper.

## 2. FULL CONTROL FLOW GRAPH

Inlining and eliminating recursive calls is to transform the code to complete the computation of the original program without using new stack frames in the user stack. Tail-recursion elimination can be regarded as a special case. In general, we basically need to deal with two issues: (1) how to expand the stack frame of the calling procedure to accommodate the local variables of inlined procedures and (2) how to realize the control flow of recursive calls without using return addresses normally saved in the stack frames of the procedures called. In this paper, we focus on the second issue and assume that all parameters of procedures are of call-by-reference as in FORTRAN and there are no local variables in procedures.

To understand what kinds of recursive calls can be inlined and eliminated and how to inline them, we need to

understand the control flow of the whole program when it is running. In this section, we define the full control flow graph of the whole program to be used in the analysis in this paper.

Procedure call and return present special control flows in the execution of the whole program. Call instruction is an unconditional jump (goto) to the first instruction of the called procedure with the return address saved in the stack frame of the calling procedure. Return instruction is another unconditional jump back to the return address saved by the corresponding call instruction. Therefore, the control flow of the whole program can be represented by the *full control flow graph* defined as follows.

Assume that there are  $n$  procedures in a program named  $P_1, \dots, P_n$ .  $CFG_i = (N_i, E_i)$  ( $1 \leq i \leq n$ ) is the control flow graph of  $P_i$ , where:

- $N_i$  is the set of the basic blocks.
- $en_i \in N_i$  and  $ex_i \in N_i$  are the *entry* and *exit* nodes of the procedure, respectively.
- Each call site in  $P_i$  is a separate basic block by itself. The call site in  $P_i$  at location  $k$  which calls procedure  $P_j$  is denoted as  $cs_{i,k,j} \in N_i$ .
- $E_i$  is the set of edges of control flow among the basic blocks in  $N_i$ .

To find the full control flow graph of the program, we first replace each call site  $cs_{i,k,j}$  with two basic blocks called *call location* and *return point*, denoted as  $cl_{i,k,j}$  and  $rp_{i,k,j}$ , respectively. We make the predecessors of  $cs_{i,k,j}$  the predecessors of  $cl_{i,k,j}$  and the successor of  $cs_{i,k,j}$  the successor of  $rp_{i,k,j}$ . Finally, we create new edges from  $cl_{i,k,j}$  to  $en_j$  called *call edge* and from  $ex_j$  to  $rp_{i,k,j}$  called *return edge*. There is no edge from  $cl_{i,k,j}$  to  $rp_{i,k,j}$ . The formal definition of full control flow graph can be found in [11].

## 3. ELIMINATABLE RECURSIVE CALLS

To eliminate recursive calls through inlining, the inlined code should realize the full control flow of the whole program without using the return addresses saved dynamically in the stack. Hence, we define a call site to be *completely inlinable* as follows:

**DEFINITION 1.** *A call site to a procedure called recursively is completely inlinable if the full control flow of the procedure calls involved can be realized without using a stack to save the orders of the calls.*

Not all call sites to procedures called recursively are completely inlinable.

Figure 3 shows a Fortran program with main procedure  $M$  and procedure  $P$ .  $b1, \dots, b4$  are the basic blocks in procedure  $P$ . Figure 4 shows its full control flow graph.

There are three edges from exit node  $ex_P$ : one to the return point  $rp_{M,1,P}$  in  $M$  and the other two to the return points,  $rp_{P,1,P}$  and  $rp_{P,2,P}$  in procedure  $P$  itself. According to the semantics of procedure call and return, the targets of the branch at the end of  $ex_P$  are determined by the order of the calls of  $P$  in the first-call-last-return fashion. While the call of  $P$  by  $M$  is known to be the first call, the order of the calls from the two call sites from  $P$  are not known until the run time. Therefore, the branch at the end of  $ex_P$

```

subroutine P(...)
  b1
  if (...) then
1:   call P(...)
    b2
  else
    b3
    if (...) then
2:   call P(...)
    b4
    endif
  endif
end

program M
1: call P(...)
end

```

Figure 3: Example 1

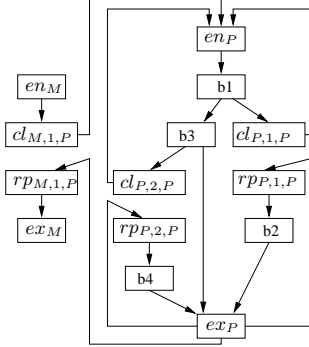


Figure 4: Full Control Flow Graph of Example 1

cannot be implemented without using a stack to save the order of the calls from  $P$  at run-time. According to Definition 1, the call site of  $P$  in  $M$  is not completely inlinable. Note that there are two back edges<sup>1</sup> to  $en_P$ , ( $cl_{P,1,P}, en_P$ ) and ( $cl_{P,2,P}, en_P$ ), in the full control flow graph in Figure 4 because  $en_P$  dominates<sup>2</sup>  $cl_{P,1,P}$  and  $cl_{P,2,P}$ . Each of them forms a natural loop<sup>3</sup> with  $en_P$  as its header. In general, if the entry node of a procedure is the header of more than one natural loops in the full control flow graph, any call site to call the procedure is not completely inlinable. This is because the branch at the end of the corresponding exit node cannot be implemented without using a stack.

Figure 5 shows another example where the call site in  $M$  to procedure  $P$  is not completely inlinable either. Figure 6 shows the full control flow graph of the program. This time, the branch at the end of  $en_Q$  is determined by the order of the two calls in  $P$  at run-time in the first-call-last-return fashion. It cannot be implemented without saving the order of the calls in a stack. Note that the natural loop with header  $en_P$

$$\{en_P, b1, cl_{P,1,Q}, cl_{P,2,Q}, en_Q, b4, cl_{Q,1,P}\}$$

contains two edges to  $en_Q$ . We can clone procedure  $Q$  and make two procedures of  $Q$ , but then  $en_P$  would become the

<sup>1</sup>A edge  $(a, b)$  is a back edge if and only if  $b$  dominates  $a$  [12].

<sup>2</sup>Node  $b$  dominates node  $a$  if and only if every path from the start node (node  $en_M$  in our case) to  $a$  goes through  $b$  [12].

<sup>3</sup>Given a back edge  $(a, b)$  in a directed graph, a natural loop with header  $b$  is all the nodes in the graph that can reach  $b$  without going through  $b$ . Node  $b$  is the header of the natural loop [12].

header of two natural loops, making the implementation of the branch at the end of  $ex_P$  impossible without using a stack. Therefore, the call site of  $P$  in  $M$  is not completely inlinable.

```

program M
1: call P(...)
end

subroutine P(...)
  b1
  if (...) then
1:   call Q(...)
    b2
  else
2:   call Q(...)
    b3
  endif
end

subroutine Q(...)
  b4
  if (...) then
1:   call P(...)
    b5
  endif
end

```

Figure 5: Example 2

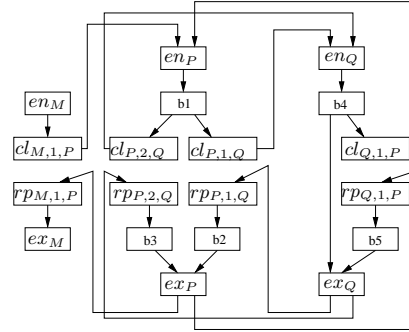


Figure 6: Full Control Flow Graph of Example 2

The situation will be completely different if the entry node of the procedure called recursively is the header of an only natural loop and any other entry node in the natural loop has only one incoming edge. Figure 7 shows such as a program. Figure 8 shows its full control flow graph. There are two edges from exit node  $ex_P$ : one is to  $rpm_{1,X}$  corresponding to the call edge  $(cl_{M,1,X}, en_X)$  and the other to  $ropy_{1,X}$  corresponding to the call edge  $(cl_{Y,1,X}, en_X)$ . Note that  $(cl_{Y,1,X}, en_X)$  is the only back edge to  $en_X$ . The non-back call edge  $((cl_{M,1,X}, en_X)$  in our case) is always first traversed before the back edge. Therefore, its corresponding return edge  $((ex_X, rpm_{1,X})$  in our case) should be traversed last after all the traversals of the other return edge  $((ex_X, ropy_{1,X})$  in our case).

```

program M
  b1
1: call X(...)
end

subroutine X(...)
  b2
  if (...) then
1:   call Y(...)
  endif
end

subroutine Y(...)
  b3
  if (...) then
1:   call X(...)
    b4
  endif
end

```

Figure 7: Example 3

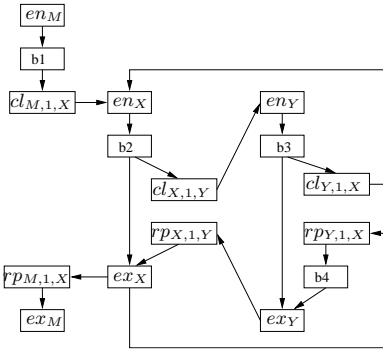


Figure 8: Full Control Flow Graph of Example 3

As can be seen from the full control flow graph of example 3 in Figure 8, the recursive call causes two loops in the full control flow graph. The first is the natural loop with the entry node of the procedure called recursively as its header. In Figure 8, it is the natural loop

$$\{en_X, b2, cl_{X,1,Y}, en_Y, b3, cl_{Y,1,X}\}$$

with header  $en_X$ . The second loop is the *dual loop* of the first loop. In Figure 8, it is

$$\{ex_X, rp_{Y,1,X}, b4, ex_Y, rp_{X,1,Y}\}$$

with  $ex_X$  as its header (in the sense of post-dominance). The number of trips of the first natural loop is determined by the conditionals in **b2** and **b3** in the loop at run-time. It is reflected in the number of times  $en_X$  is visited. The control flow of the back edge ( $cl_{Y,1,X}, en_X$ ) in our case can be simply realized by a jump instruction. The number of trips of the dual loop can only be controlled by the branch at the end of  $ex_X$ . The number of visits of  $ex_X$  should equal to that of  $en_X$ . To control the dual loop and implement the branch at the end of  $ex_X$ , we can introduce a special variable called  $vc_X$  (visit-count) to record the number of times the entry node  $en_X$  is visited. In the code of  $cl_{M,1,X}$ , we insert the code to initialize  $vc_X$  to zero:  $vc_x \leftarrow 0$ . In the code of  $en_X$ , we insert the code to increment it:  $vc_x \leftarrow vc_x + 1$ . Since the call of  $X$  by  $M$  is known to be the first one, its corresponding return to  $rp_{M,1,X}$  should be the last one. Since there is only one destination for other returns, namely to  $rp_{M,1,X}$ , the control of the branch at the end of  $ex_X$  can be realized by the following code:

```

vc_X ← vc_X - 1;
if (vc_X > 0) then
    goto rp_{Y,1,X}
else
    goto rp_{M,1,X}
endif

```

Note that the entry node of procedure  $Y$ ,  $en_Y$ , has only one incoming edge. Hence, its exit node,  $ex_Y$ , has only one destination. It is easy to implement it by simply inlining  $Y$  in  $X$ . Therefore, the call of  $X$  in  $M$  of this example is completely inlinable.

We can see that a call site of a procedure called recursively can be completely inlined if and only if the entry node of the procedure called is the header of an only natural loop in the full control flow graph and any other entry node in the

natural loop has only one incoming edge. Such a natural loop is called a *simple loop* in this paper.

We have been using full control flow graph to understand the actual control flow of the whole program and why and how a call site to the header of a simple loop in the full control flow graph can be completely inlined. The full control flow graph of the whole program can be very large. Using the full control flow graph to find the completely inlinable call sites would be expensive. Fortunately, we do not have to use the full control flow graph for this purpose.

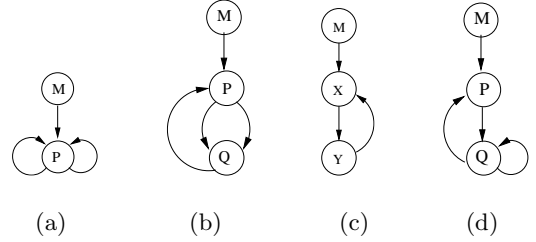


Figure 9: Call Graphs

The traditional call graph [5, 10] is a multi-graph where each node represents a procedure in the program and each edge, say  $(P, Q)$ , represents a call site in procedure  $P$  which invokes procedure  $Q$ . Figures 9(a), 9(b) and 9(c) show the call graphs of the programs in Figures 3, 5 and 7, respectively. In essence, the call graph provides the flow-insensitive information about call sites in the whole program. We can similarly define a *simple loop* in a call graph to be a natural loop whose header is not the header of any other natural loop and each of its other nodes has only one incoming edge. We can prove that the entry node of a procedure in the full control flow graph is the header of a simple loop if and only if the corresponding node of the procedure is the header of a simple loop in the control flow graph. Therefore, any call site corresponding to a non-back edge to the header of a simple loop in the call graph is completely inlinable. In Figure 9(a), the two natural loops involving  $P$  are not simple loops, because they share  $P$  as their headers. The natural loop  $\{P, Q\}$  with header  $P$  in Figure 9(b) is not a simple loop either, because  $Q$  has two incoming edges. The call site of  $P$  in  $M$  in either program is not completely inlinable. The natural loop  $\{X, Y\}$  in Figure 9(c) is a simple loop with header  $X$ . The call site of  $X$  in  $M$  is completely inlinable.

## 4. COMPLETE INLINING

The loops caused by recursive calls may be nested. In the call graph of Figure 9(d) (whose code is not shown and can be found in [11]), the simple loop  $\{Q\}$  is nested in the natural loop  $\{P, Q\}$ . After the call site  $(P, Q)$  is completely inlined by using the algorithm to be described in this section, the call site  $(Q, Q)$  is eliminated. The simple loop  $\{Q\}$  will be collapsed to a single node  $Q$ , and the outer natural loop  $(P, Q)$  becomes a simple loop. Compilers can find such nested loops in the call graph and perform complete inlining from the outermost loop to the innermost one.

Figure 10 shows the inlining algorithm to completely inline call site  $(m, n_1)$  in procedure  $m$  where  $n_1$  is the header of a simple loop  $L = (n_1, \dots, n_p)$  in the call graph. Let the

```

procedure inline( $m, n_1, \dots, n_p$ );
    //line the simple loop ( $n_1, \dots, n_p$ )
    //    at call site ( $m, n_1$ ) in  $m$ 
begin
(1)  remove call site ( $m, n_1$ );
(2)  insert code for the call location:
(3)       $vc\_n1 := 0$ 
(4)      ... //code to pass arguments
(5)  insert code for entry of  $n_1$ :
(6)       $en\_n1: vc\_n1 := vc\_n1 + 1$ 
(7)  for  $k = 1$  to  $p - 1$  do
(8)      inline the code of  $n_k$  as usual;
(9)  endfor
(10) inline the code of  $n_p$  and do
(11)   replace the call site ( $n_p, n_1$ ) in  $n_p$  with:
(12)     ... //code to pass arguments
(13)     goto  $en\_n1$ 
(14)    $rp$ : ... //code to return parameters
(15) enddo
(16) insert code for exit of  $n_1$ :
(17)    $vc\_n1 := vc\_n1 - 1$ 
(18)   if ( $vc\_n1 > 0$ ) then
(19)     goto  $rp$ 
(20)   endif
(21)   ... //code to return parameters
end

```

**Figure 10: Inlining Algorithm**

call site  $(m, n_1)$  in procedure  $m$  be represented by  $cs_{m,1,n_1}$  using the notation in Section 2. The call site is to be replaced by the codes for its call location and return point:  $cl_{m,1,n_1}$  and  $rp_{m,1,n_1}$ . Since  $n_1$  is the header of a simple loop, a visit-count variable,  $vc\_n1$ , needs to be introduced to control the exit branch of  $n_1$  as described in Section 3. The first instruction for  $cl_{m,1,n_1}$  is to initialize  $vc\_n1$  to zero. It is followed by the code to bind the arguments to the formal parameters for  $cs_{m,1,n_1}$ . The last instruction of  $cl_{m,1,n_1}$  is supposed to be a jump to the entry of  $n_1$  according to the full control flow graph described in Section 2. But, since we are going to inline the body of  $n_1$  at the current point, the jump instruction is not necessary. The code to be inserted for  $cl_{m,1,n_1}$  are lines (3)-(4) in the procedure inline(). Next, the algorithm inserts the code for the entry node of  $n_1$  which is the increment of  $vc\_n1$  with address label  $en\_n1$  as shown at line (6). After this, the algorithm inlines the body of  $n_1$  and the bodies of  $n_2, \dots, n_{p-1}$  using the traditional inlining technique (copy the code, pass or change the parameter names to be the argument names, etc) as shown in lines (7)-(9). Lines (10)-(15) are to inline  $n_p$  in  $n_{p-1}$  and eliminate the recursive call site  $(n_p, n_1)$  in  $n_p$ . In particular, the call site  $(n_p, n_1)$ , denoted as  $cs_{n_p,1,n_1}$ , is to be replaced by the call location  $cl_{n_p,1,n_1}$  and the return point  $rp_{n_p,1,n_1}$ . The code to be inserted for  $cl_{n_p,1,n_1}$  is to pass the arguments to the parameters for  $cs_{n_p,1,n_1}$  and then jump to  $en\_n1$ . The code for  $rp_{n_p,1,n_1}$  is the instructions to return the parameters to the arguments for the call site. It has address label  $rp$  as the target for the jump instruction from the exit node of  $n_1$ . Lines (16)-(21) are to insert the code for the exit node of  $n_1$ . In particular, the code to be inserted

is to decrement  $vc\_n1$  and make a conditional branch to the return point  $rp_{n_p,1,n_1}$ , namely  $rp$ , as described in Section 3. The conditional branch to the return point  $rp_{m,1,n_1}$  is not necessary, because  $n_1$  is inlined here at the right position. Finally, we need to insert the code to return the parameters to the arguments for  $cs_{m,1,n_1}$  at this position (line (21)).

To implement the binding of the arguments to the call-by-reference parameters at a call site for inlining, we first keep the names of all the parameters and adopt the binding abstraction used in the Program Summary Graph [13]. That is, the argument and the corresponding parameter are used and defined, respectively, at the call location, and defined and used, respectively, at the return point. Therefore, for each argument  $A$  bound to parameter  $X$  at a call site, there is an assignment  $X := A$  in the code of the call location and an assignment  $A := X$  in the code of the return point. Consider the program in Figure 11(a). Figure 11(b) shows its completely inlined program using the inlining algorithm in Figure 10. Note that the names of parameters  $a$  and  $b$  are used in the main program. At the call site in the main program in Figure 11(a),  $c$  and  $d$  are bound to  $a$  and  $b$ , respectively. But, at the call site in  $X$ ,  $b$  and  $a$  are bound to  $a$  and  $b$ , respectively. Note that we need to use variable  $tmp$  in the assignments at the call location and the return point of the call site in  $X$ . After the inlining is completed, we can eliminate all the parameters by substituting them with the arguments of the call site inlined. Figure 11(c) shows the code after all  $a$  and  $b$  are substituted with  $c$  and  $d$ , respectively.

However, if the recursive call uses the same parameters of the procedure as its arguments as in Figure 1, the binding assignments can be eliminated. All the parameters can be eliminated as shown in Figure 2(a).

## 5. CONCLUSION AND RELATED WORK

We have presented a novel compiler optimizing transformation called complete inlining to inline and eliminate recursive calls in programs to either eliminate the cost of procedure calls or to facilitate further compiler analysis and optimization for high-performance and parallelization. Our transformation inlines and eliminates the recursive calls that cannot be eliminated either by tail-recursion elimination [5, 6] or the current inlining [1, 2, 3].

After recursive calls are inlined and eliminated, all data flow, control flow and memory use information of the procedures are fully exposed. This opens the door for further compiler analysis and optimization including induction variable analysis [7, 8, 9]. While some interprocedural induction variables can be discovered by interprocedural induction variable analysis [14], the induction variable analysis after the recursive calls are completely inlined is less costly. The induction variable analysis after the complete inlining can discover the induction variables in nested loops which may be difficult to find by using interprocedural induction variable analysis [14].

This work builds on the concept of the full control flow of the whole program. The full control flow graph (FCFG) defined in this paper is related to various full program representations such as Program Summary Graph (PSG) [13], Full Program Representation Graph (FPR) [15] and Extended Full Program Representation Graph (EFPR) [14]. The full control flow graph can be regarded as the complete representation of all control flows in the whole program, al-

<pre> program main   c = 5   d = 2   call X(c,d)   print *, c,d end  subroutine X(a,b)   a = a - b   if (a &gt; b) then     call X(b,a)     b = 2 * a   endif end </pre> <p>(a) Original Code</p>	<pre> program main   c = 5   d = 2   vcX = 0   a = c   b = d enP: vcX = vcX + 1   a = a - b   if (a &gt; b) then     tmp = a     a = b     b = tmp     goto enP rp:   tmp = a   a = b   b = tmp   b = 2 * a   endif   vcX = vcX - 1   if (vcX &gt; 0) then     goto rp   endif   c = a   d = b   print *, c,d end </pre> <p>(b) Inlined Code</p>	<pre> program main   c = 5   d = 2   vcX = 0 enP: vcX = vcX + 1   c = c - d   if (c &gt; d) then     tmp = c     c = d     d = tmp     goto enP rp:   tmp = c   c = d   d = tmp   d = 2 * c   endif   vcX = vcX - 1   if (vcX &gt; 0) then     goto rp   endif   print *, c,d end </pre> <p>(c) Final code</p>
---	--	--

Figure 11: Example 4

though it may not be realized without using stacks. The other graphs reflect certain abstractions of the full control flow graph for the analyses for which they are intended [13, 15, 14].

## 6. REFERENCES

- [1] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Ruskin, editor, *Design and Implementation of Compilers*. Prentice-Hall, 1971.
- [2] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 134–145, 1997.
- [3] Jan Hubicka. The gcc call graph module: A framework for inter-procedural optimization. In *Proceedings of the GCC Developers' Summit*, pages 65–78, June 2004.
- [4] Xavier Vera and Jingling Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 175–186, February 2002.
- [5] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
- [6] Andreas Bauer. Compilation of Functional Programming Languages using GCC – Tail Calls. Master's thesis, Department of Informatics, Munich University of Technology, Munich, Germany, 2003.
- [7] Shin-Ming Liu, Raymond Lo, and Fred Chow. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the IEEE 1996 Conference on Parallel Architectures and Compilation Techniques*, pages 228–237, 1996.
- [8] S. Pop, P. Claus, V. Loechner, and G.-A. Silber. Fast recognition of scalar evolutions on three-address SSA code. Technical Report A/354/CRI, Centre de Recherche en Informatique (CRI), 2004.
- [9] R. van Engelen, J. Birch, Y. Shou, B. Wash, and K. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the ACM 18th International Conference on Supercomputing*, pages 106–115, 2004.
- [10] Ken Kennedy and Randy Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2002.
- [11] Peiyi Tang. Complete inlining of recursive calls: Beyond tail-recursion elimination. Technical Report titus.compsci.ualr.edu/~ptang/papers/btr.pdf, Department of Computer Science, University of Arkansas at Little Rock, 2005.
- [12] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, 1986.
- [13] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–56, June 1988.
- [14] Peiyi Tang and Pen-Chung Yew. Interprocedural induction variable analysis. *International Journal of Foundations of Computer Science*, 14(3):405–423, June 2003.
- [15] 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.