Efficient Graph Call Analysis

Mary Hall
Ken Kennedy

CRPC-TR92223
July, 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Efficient Call Graph Analysis*

Mary W. Hall
Center for Integrated Systems
Stanford University
Stanford, CA 94305

Ken Kennedy
Department of Computer Science
Rice University
Houston, TX 77251

July 10, 1992

Abstract

This paper presents an efficient algorithm for computing the procedure call graph, the program representation underlying most interprocedural optimization techniques. The algorithm computes the possible bindings of procedure variables in languages where such variables only receive their values through parameter passing, such as Fortran. We extend the algorithm to accommodate a limited form of assignments to procedure variables. The resulting algorithm can also be used in analysis of functional programs that have been converted to Continuation Passing Style.

1 Introduction

Although most compilers optimize procedures as separate units, there is increasing evidence that optimization across procedure boundaries can yield significant improvements in program execution times. Interprocedural analysis and optimization have proven to be important to automatic parallelization of loops containing procedure calls [TIF86, LY88, EB91, HK91, HKM91, McK92], enhancing memory hierarchy performance [BCHT90, McK92] and compiling for distributed-memory multiprocessors [HHKT91]. The above research focuses on analyzing languages used by scientific programmers, usually Fortran. However, interprocedural optimization is perhaps even more important for functional languages, where functions are small and calls occur quite frequently. This idea inspired research in applying traditional data-flow analysis to functional languages; in this context, data-flow analysis must be formulated as an interprocedural problem [Shi91a].

Any technique performing analysis or optimization across procedure boundaries requires some underlying representation of the program structure. Most often the structure used is the *call graph*, a static structure describing the dynamic invocation relationships between procedures in a program. A node in the call graph represents a procedure (or function), and an edge $(p \rightarrow q)$ exists if procedure p can invoke procedure q. Note that the call graph is actually a multigraph since each

*This research has been supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center, by IBM Corporation and by the state of Texas.

static invocation from p to q is represented by a distinct edge; for historical reasons, we refer to it as a graph.

The approach to building the call graph depends on the language being compiled. When all invoked procedures are *statically bound* to procedure constants, the call graph is constructed in a single sweep over the procedures and call sites in the program. For languages where *dynamically bound* calls can occur, the compiler must perform analysis to determine all possible bindings for invoked procedures. We call this problem *call graph analysis*. Dynamically bound calls result from a number of language features including the parameter passing mechanism and variable assignment. In this paper, we focus on the first case, supporting languages where dynamically bound calls arise from invocations of procedure-valued formal parameters that only receive their values via parameter passing. We extend the algorithm to provide a simple solution when assignment to procedure-valued variables is allowed.

Our algorithm produces essentially the same call graph described by other less efficient algorithms [Spi71, Wal76, Wei80, Bur87, Shi88]. More precise algorithms exist, but they may be prohibitively expensive depending on the language features [Ryd79, CCHK90, Shi91b]. While most of these approaches were designed for imperative languages, Shivers proposed an approach for analyzing control flow of Scheme in Continuation Passing Style (CPS) form.

The paper is organized into nine remaining sections and a conclusion. The next section explains how CPS-conversion can be used to make the algorithm applicable to a wider variety of languages. Section 3 provides a description of the call graph analysis problem. Section 4 presents the algorithm, and Section 5 gives an example that exercises all of its steps. Section 6 proves the algorithm is correct, and Section 7 presents its time complexity. Section 8 briefly describes an implementation of the algorithm. Section 9 shows the extensions needed to handle assignment to procedure variables. Section 10 discusses previous work, including an in-depth comparison with the precise algorithms.

2 Continuation Passing Style and Analyzing Scheme

Continuation Passing Style provides an intermediate program representation where all transfers of control are represented by tail recursive procedure calls [Fis72, Rey72]. A procedure is passed a *continuation* as one of its parameters. A continuation is another procedure representing the position in the code in which to transfer control; rather than returning, a procedure simply invokes its continuation on exit.

Shivers provides a formulation of control flow analysis in Scheme, analyzing a program after CPS conversion [Shi88]. CPS conversion eliminates procedure-valued returns, replacing them with invocations to a formal parameter. The call graph analysis algorithm presented in this paper, while designed to analyze languages where the only dynamically bound procedure calls are through procedure-valued formal parameters, can also analyze languages with procedure-valued returns after CPS conversion.

This point is illustrated with the following example. Here, assume that the language has call-by-value parameter-passing semantics.

```
program main            function a(f)          procedure c(f)
    call c(a(print))        return f               call f(1)
```

In this program, the function call *a(print)* is evaluated, and the result is passed to *c*. The return value is the procedure *print*, which *c* subsequently invokes. A CPS form for this code segment is as follows.

```
program main              procedure a(f, g)      procedure m-c (f)      procedure c(f)
    call a(print, m-c)        call g(f)              call c(f, K)           call f(1, g)
```

We have added the procedure *m-c* to represent the control flow following the call to *a*. The function *a* is now a procedure that passes its return value to its continuation. The continuation invokes *c* with this parameter and passes the top continuation K to *c*, which *c* passes along to the function it invokes. In this form, determining the value of *c*'s invoked formal *f* (and the continuations for *a* and *c*) involves simple analysis of the parameter passing in the program.

The observation that CPS conversion permits call graph analysis to consider only parameter passing of procedure values is due to Shivers. Once the connection between the two problems was understood, we became interested in applying our algorithm to analyzing Scheme. However, there were some additional issues to be addressed. First, Shivers allows assignments to data structures, so it is possible to store a function in a data structure and later retrieve and invoke it. We provide a simple extension to the algorithm to accommodate assignments in Section 9, which is at least as precise as what Shivers suggests. Second, he analyzes incomplete programs where some functions can be invoked by the external environment and invocations appear to external functions. We ignore this issue in the rest of the paper. It should not be any more difficult to deal with incomplete programs as long as we can determine which functions can be invoked by the external environment.

3 Simultaneous Equations

We can determine the structure of the call graph by simulating the parameter passing during execution. For a given procedure formal pf, we calculate the set *Boundto(pf)*, the procedure constant bindings for pf. For some call site invoking formal pf, *Boundto(pf)* is the set of procedures that may be invoked at the call. Let *pc* represent a procedure constant and pf_p a procedure formal of procedure p. Then, the *Boundto* sets can be described with the following simultaneous equations.

$$Boundto(pc) = \{pc\}$$
$$Boundto(pf_p) = \cup_{c\ invokes\ p}\ Boundto(Passed'(pf_p, c))$$

The above equations indicate that a procedure constant has itself as its only binding. A procedure formal receives bindings from the actual parameters passed to it at calls invoking the procedure in which it appears. The function *Passed* provides the procedure formal or procedure constant passed to pf_p at call site *c*. *Passed'* is a slight modification to *Passed* needed to improve the precision of the algorithm. The equation for *Passed'* is as follows.

$Passed'(pf_p, c) =$

$$\begin{cases} \{p\} & \text{if } pf_q \textit{ is invoked at } c \wedge Passed(pf_p, c) = pf_q \\ Passed(pf_p, c) & \text{otherwise} \end{cases}$$

To understand the difference between *Passed* and *Passed'*, consider a procedure parameter pf_q with multiple values in its *Boundto* set, one of which is p. If pf_q is invoked at call c and also appears as the actual parameter passed to pf_p, *Boundto(Passed(*pf_p, c*))* would include all of the values in *Boundto(*pf_q*)*. However, we know that p can only be invoked at this call when pf_q has binding p, so we are introducing imprecision by propagating bindings other than p to pf_p.

These simultaneous equations could easily be formulated into a simple iterative algorithm for call graph analysis, but the resulting solution requires reanalysis on the graph each time a new binding is located for an invoked procedure variable (*i.e.*, a new edge is added to the call graph). Several algorithms presented in Section 10 suffer from this inefficiency. Instead, our algorithm propagates new bindings on demand as they are located.

4 Algorithm

The algorithm for call graph analysis is given in Figure 1. The procedure *InitializeNode* is called on a newly reachable procedure to initialize the *Boundto* sets for its procedure parameters and to locate entries for *Worklist*. It recursively calls itself to initialize procedures that become reachable through statically bound edges. By initializing procedures only as they become reachable, we ensure that procedure parameter bindings result only from reachable call chains.

The main algorithm *Build* relies on a list of pairs *Worklist*. A pair $\langle a, pf_p \rangle$ is in *Worklist* if a can be bound to pf_p through parameter passing along some chain of calls and a is possibly not yet in *Boundto(*pf_p*)*. Initially, *Worklist* contains those pairs representing bindings from statically bound calls; that is, a would be the actual parameter passed to pf_p at some call to p. The job of *Build* is to propagate bindings for procedure formals at statically bound calls, and handle both procedure constants and procedure formals at dynamically bound calls.

When a pair $\langle a, pf_p \rangle$ is removed from *Worklist*, we must examine all call sites in procedure p, looking for uses of pf_p. There are three distinct situations that may arise at each call site.

1. **The call site is statically bound.**
   Then, pf_p may appear as an actual at the call; the binding a is propagated to the corresponding parameter of the called procedure (step 1 of the algorithm).
2. **The call site is dynamically bound, but invokes some formal other than pf_p.**
   Again, pf_p may appear as an actual at the call; the binding a is propagated to the corresponding parameter of all procedures currently bound to the invoked formal (step 1).
3. **The call site invokes pf_p.**
   We propagate procedure constants at the call to the corresponding formals of a (step 2b). We also propagate known bindings for the other procedure formals appearing at the call to the

```
/* Add a reachable node and all of its statically bound edges */
procedure $InitializeNode(p)$
      add $p$ to $Nodes$
      foreach formal procedure parameter $pf_p$
          $Boundto(pf_p) \leftarrow \emptyset$
      foreach call site $c = (p \rightarrow q)$ in $p$
          if $q$ is a procedure constant then
             if $q \notin Nodes$ then call $InitializeNode(q)$
             add edge $(p \rightarrow q)$ to call graph
             foreach procedure constant $pc \in WalkConstants(c)$
                foreach $pf_q$ in $isPassed(pc, c, q)$
                   add $\langle a, pf_q \rangle$ to $Worklist$
end /* InitializeNode */


/* Build call graph starting at root procedure */
program Build
     $Worklist \leftarrow \emptyset$
     $Nodes \leftarrow \emptyset$
     call InitializeNode($main$)

     while $Worklist \neq \emptyset$
          select and delete a pair $\langle a, pf_p \rangle$ from $Worklist$
          if $a \notin Boundto(pf_p)$ then
             add $a$ to $Boundto(pf_p)$
             foreach call site $c = (p \rightarrow q)$ in $p$
(1)             if $q \neq pf_p$ then
                   foreach $b \in Boundto(q)$
                      foreach $pf_b \in isPassed(pf_p, c, b)$
                         add $\langle a, pf_b \rangle$ to $Worklist$

(2)             else /* $pf_p$ is invoked at the call */
                   if $a \notin Nodes$ then call $InitializeNode(a)$
                   add edge $(p \rightarrow a)$ to call graph
                   foreach procedure-valued actual $pa \in WalkActuals(c)$
(2a)                  if $(pa = pf_p)$ then
                         foreach $pf_a \in isPassed(pf_p, c, a)$
                            add $\langle a, pf_a \rangle$ to $Worklist$
(2b)                  else
                         foreach procedure constant $pc \in Boundto(pa)$
                            foreach $pf_a \in isPassed(pa, c, a)$
                               add $\langle pc, pf_a \rangle$ to $Worklist$
end /* Build */
```

Figure 1 Algorithm for call graph analysis.

corresponding parameters of a (step 2b). In the special case that pf_p appears as an actual at the call, we propagate a to the corresponding formal of a (step 2a).

Steps 1 and 2 of the algorithm complement each other. If we receive a binding b for formal pf passed at a dynamically bound call before bindings for the invocation, step 2b will propagate b when bindings for the invoked formal become available. If we receive a binding for an invoked formal before bindings for the formals passed as actuals at the call, step 1 propagates bindings for the actuals as they become available.

The algorithm relies on a few definitions. First, *isPassed* is the inverse function for *Passed.* For a procedure formal pf_p that appears in one or more actual parameter positions at call site c in p, *isPassed*(pf_p, c, b) returns the corresponding formal parameters of procedure b. If c is a dynamically bound call, then b is in the *Boundto* set of the invoked procedure formal. For convenience, we also define two iterator functions.

WalkConstants(c): the set of procedure constants passed as actual parameters at call c

WalkActuals(c): the set of procedure-valued actual parameters passed at call c

5 An Example

This section works through the steps of the algorithm to make them clearer. Consider the following program.

```
program main          procedure a(f1, f2)      procedure b(f3, f4)
    call a(b, c)          call b(a, f2)            call f3(f3, d)
                          call f1(f1, f2)          call f4
```

The initialization step locates the passing of procedure constants in *main* at the call to a, and further, in a at the call to b. The initial *Worklist* entries are $\{\langle b, f_1\rangle, \langle c, f_2\rangle, \langle a, f_3\rangle\}$, and the initial edges are $(main \rightarrow a)$ and $(a \rightarrow b)$.

Now the pairs on *Worklist* are processed. The steps are as shown in Figure 2. Each *Worklist* entry is annotated with the step of the algorithm that caused it to be added. After the last step shown in Figure 2, the remaining pairs on *Worklist* are $\{\langle b, f_3\rangle, \langle d, f_4\rangle, \langle a, f_1\rangle, \langle c, f_2\rangle, \langle d, f_2\rangle\}$, all of which have already been processed by the algorithm. The algorithm terminates with the final values for *Boundto* and the resulting call graph as shown in Figure 3.

6 Proof of Correctness

Lemma 1 *The algorithm Build terminates after no more than c_pEP iterations of the while loop.*

Proof. Each iteration of the *while* loop removes a pair from *Worklist*. Each addition to *Worklist* represents the propagation of a new binding of a formal procedure parameter along a distinct edge. Let E be the number of edges in the final call graph. (We refer to an edge rather than a call since a dynamically bound call generates a distinct edge for each of its bindings.)

```
Process $\langle b, f_1\rangle$:
    $Boundto(f_1) \leftarrow \{b\}$
    add edge $(a \rightarrow b)$ to call graph
    add $\langle b, f_3\rangle$ to $Worklist$ (1)

Process $\langle c, f_2\rangle$:
    $Boundto(f_2) \leftarrow \{c\}$
    add $\langle c, f4\rangle$ to $Worklist$
        (really added twice) (1)

Process $\langle a, f_3\rangle$:
    $Boundto(f_3) \leftarrow \{a\}$
    add edge $(b \rightarrow a)$ to call graph
    add $\langle d, f_2\rangle$ to $Worklist$ (2b)
    add $\langle a, f_1\rangle$ to $Worklist$ (2a)

Process $\langle b, f_3\rangle$:
    $Boundto(f_3) \leftarrow \{a, b\}$
    add edge $(b \rightarrow b)$ to call graph
    add $\langle d, f_4\rangle$ to $Worklist$ (2b)
    add $\langle b, f_3\rangle$ to $Worklist$ (2a)

Process $\langle c, f_4\rangle$:
    $Boundto(f_4) \leftarrow \{c\}$
    add edge $(b \rightarrow c)$ to call graph

Process $\langle d, f_2\rangle$:
    $Boundto(f_2) \leftarrow \{c, d\}$
    add $\langle d, f_4\rangle$ to $Worklist$
        (added twice) (1)

Process $\langle a, f_1\rangle$:
    $Boundto(f_1) \leftarrow \{b, a\}$
    add edge $(a \rightarrow a)$ to call graph
    add $\langle a, f_1\rangle$ to $Worklist$ (2a)
    add $\langle c, f_2\rangle$ to $Worklist$ (2b)
    add $\langle d, f_2\rangle$ to $Worklist$ (2b)

Process $\langle d, f_4\rangle$:
    $Boundto(f_4) \leftarrow \{c, d\}$
    add edge $(b \rightarrow d)$ to call graph
```

Figure 2 Steps of algorithm for example program.

For each edge, the maximum number of pairs added to *Worklist* is equal to the number of unique bindings of the procedure formals appearing as actual parameters at this call. Let P be the number of unique procedure constants passed as parameters in the program. If we assume a procedure has at most c_p parameters, where c_p is some small constant, then the number of pairs added to *Worklist* as a result of a single call site is $O(c_pP)$. Thus, the number of pairs added to *Worklist* and the number of iterations of the *while* loop are bounded by $O(c_pEP)$.

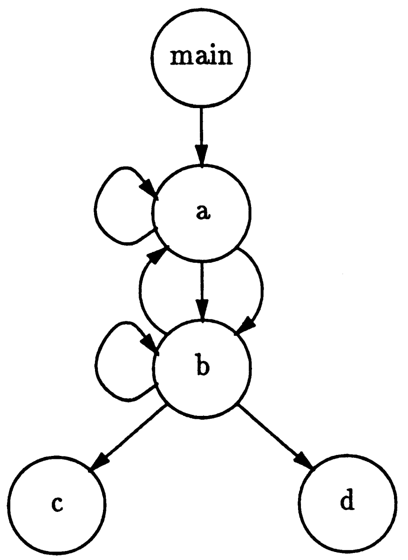
Lemma 2 *The algorithm Build constructs the portion of the static call graph reachable from the root node through statically bound calls before entering the while loop.*

The proof is obvious from the definition of *InitializeNode*.

Lemma 3 *The algorithm adds a pair $\langle a, pf_{n_m}\rangle$ to Worklist if and only if through some chain of calls $n_0 \xrightarrow{c_1} n_1 \xrightarrow{c_2} \ldots \xrightarrow{c_m} n_m$, a is bound to pf_{n_m} at c_m.*

Proof.
$\Rightarrow$: By induction on the *while* loop iteration count.

Basis. By Lemma 2, on entry to the *while* loop *Worklist* contains the set of pairs $\langle a, pf_p\rangle$ such that procedure constant a is passed directly to pf_p at some statically bound call site invoking p.



$Boundto(f_1) = \{b, a\}$
$Boundto(f_2) = \{c, d\}$
$Boundto(f_3) = \{a, b\}$
$Boundto(f_4) = \{c, d\}$

Figure 3 Resulting call graph and *Boundto* sets from example.

Induction. Assume the lemma is true after the first n iterations of the loop. The pair $\langle a, pf_p \rangle$ selected on iteration $n + 1$ must then be an element in the final set *Boundto(pf_p)*. If the binding has already been propagated, we ignore this pair and continue to the first iteration that contributes a new binding.

The proof follows from considering what *Worklist* pairs are generated by the new binding. We visit each call site in p such that pf_p is either invoked or is an actual parameter. The propagation at these calls corresponds to the rules set forth in Section 4. In each case, it can be shown that for any generated *Worklist* pair $\langle pf, b \rangle$, $b \in Boundto(pf)$ is true. (A detailed proof can be found elsewhere [Hal91].)

If pf_p is invoked, then a may have become a reachable procedure as a result of this call. In this case, *InitializeNode* will add bindings from any static calls through a. That these statically bound calls are correctly processed follows from Lemma 2.

$\Leftarrow$: By induction on the length of the call chain.

Basis. The only length-0 call chain just contains the root node, which contributes no bindings.

Induction. Assume that for every call chain $n_0 \xrightarrow{c_1} \ldots \xrightarrow{c_m} n_m$ of length m, pairs $\langle a, f_{n_m} \rangle$ are inserted into *Worklist* representing bindings for the formal parameters of n_m. Let $n_0 \xrightarrow{c_1} n_1 \xrightarrow{c_2} \ldots \xrightarrow{c_m} n_m \xrightarrow{c_{m+1}} n_{m+1}$ be some call chain in the program. By the induction hypothesis, all bindings for the procedure formals of n_m that result from this call chain have been inserted into *Worklist*.

Consider the effect of propagating bindings of n_m's formals to n_{m+1} across call c_{m+1}. If c_{m+1} is a dynamically bound call invoking some formal f_{n_m}, then one binding for f_{n_m} must be n_{m+1}. By the induction hypothesis, the pair $\langle n_{m+1}, f_{n_m} \rangle$ must have been added to *Worklist* when actuals passed at c_m were propagated to n_m.

Bindings for all other formals of n_m through this call chain arise from the procedure constants and procedure formals passed as actuals at c_{m+1}. For actuals that are procedure constants, the bindings are either added by *InitializeNode* (if c_{m+1} is statically bound to n_{m+1}) or during processing of the pair $\langle n_{m+1}, f_{n_m} \rangle$. Actuals that are procedure formals of n_m will contribute their bindings once they are processed. By the induction hypothesis, all bindings for formals of n_m have been added to *Worklist*, and by Lemma 1, will eventually be processed. □

Theorem 1 *Build correctly calculates the Boundto sets as defined in Section 3, and as a result, correctly computes the call graph for any input program.*

Proof. By Lemma 1, the algorithm terminates, and by Lemma 2, *InitializeNode* correctly builds the static portion of the call graph. Lemma 3 establishes that bindings for procedure parameters are correctly determined based on parameter passing along chains of calls in the graph. With these bindings, edges representing dynamically bound calls are correctly added to the graph. Since we have shown that both statically and dynamically bound calls are correctly added, we have the desired result. □

7 Time Complexity

Theorem 2 *The entire call graph analysis algorithm is bounded by $O(N + EP)$ time.*

Proof. The procedure *InitializeNode* examines a procedure's call sites once for each formal procedure parameter. It is invoked once for each node in the final call graph. Let c_p be the maximum number of procedure parameters of any procedure. Then the initialization step requires $O(c_p(N + E))$, where E and c_p are as before, and N is the number of nodes in the final call graph. We assume c_p is bounded by a small constant and omit it in the remaining discussion, consistent with assumptions made in other work [CK88, CK89].

The procedure *Build* processes a *Worklist* entry on each iteration of the loop. Lemma 1 asserted that the number of pairs appearing on *Worklist* is bounded by $O(EP)$. Consider the number of unique pairs on *Worklist*. As part of the processing of a pair $\langle a, pf \rangle$, the procedure constant a is added to the *Boundto* set of the procedure parameter pf. If this pair is ever selected from *Worklist* again, it will not be processed since a already appears in *Boundto(pf)*. Thus, the number of iterations of the *while* loop that proceed past the initial test is no greater than the number of possible pairs in *Worklist*. These are $\langle$ procedure constant, procedure parameter $\rangle$ pairs, so the bound is $O(NP)$. Since the number of unique *Worklist* pairs for a given procedure is bounded by P, in the *for* loop we visit a distinct call site $O(P)$ times. As a result, examining call sites takes $O(EP)$ time.

Consider the inner loops that propagate newly determined bindings of procedure parameters. Each operation inside these loops adds pair to *Worklist*. These steps are then bounded by the size of *Worklist* or $O(EP)$.

8 Implementation Results

The algorithm has been fully implemented. It provides the basis for interprocedural optimization in ParaScope, an environment designed for supporting scientific Fortran programmers [CCH+88]. The research in ParaScope has focused on aggressive compilation of scientific Fortran codes for a variety of architectures and has pioneered the incorporation of interprocedural optimization into an efficient compilation system.

The call graph algorithm was largely implemented directly from the presentation here. There is one interesting difference worthy of note. As we locate a call invoking a procedure formal during initialization, we add a dummy node to the call graph to represent the call. We add an edge from the caller to the dummy node. During propagation, when a new binding is located for the procedure formal, we add an edge from the dummy node to the procedure that is potentially invoked at the call.

This indirection has no effect on the results of interprocedural analysis, but it simplifies queries on the call graph. When an invoked procedure formal has multiple bindings, a single call through the formal corresponds to multiple edges in the graph based on the algorithm in Figure 1. Queries about the interprocedural information at the call would need to examine each edge and form the meet of the information. By adding indirection, a call with multiple bindings corresponds to a single edge from the caller to the dummy node. The queries need only examine the information at this edge.

9 Extensions for Assignment to Procedure Variables

We propose a simple approach to analyzing assignments that is at least as precise as the Shivers definition. In the following discussion, we consider assignments of the form $a = b$, where a is a procedure variable and b is either a procedure constant or a procedure variable.[1]

9.1 Extensions to the Algorithm

Let us define a set γ containing those procedure constants that may be bound to any procedure variable through assignment. γ is initialized by a preliminary examination over all the procedures in the program, adding procedure constant *pc* to γ if *pc* appears as the right-hand operand of an assignment or if it appears as an actual parameter at a call site.

Once γ has been initialized, the algorithm proceeds as before with a few minor changes. During *InitializeNode*, we need to incorporate three changes.

1. While examining the actual parameters at statically bound calls, we look for procedure variables that are not formals. We treat these as procedure constants, adding the appropriate pairs to *Worklist* for each of the bindings in γ.

[1]Shivers instead restricts assignment to data structures via primitive functions such as "cons." The above analysis can be made equivalent by considering the side effects due to such operations as the left and right sides of an assignment.

2. We look for dynamically bound calls invoking procedure variables that are not formals. We treat the invocations as statically bound calls to each of the procedure constants in γ, adding the appropriate *Worklist* entries as dictated by the initialization algorithm.
3. We examine the procedure for assignments to procedure-valued formal parameters. These formals may receive their values either from parameter passing or assignment. When such a procedure formal pf is located, we add the pairs $\langle pc, pf \rangle$ to *Worklist* for each of the procedure constants in γ.

During propagation, we make the following two changes.

1. When iterating over the calls in *Build*, if the call invokes a procedure variable q that is not a formal, we assume $Boundto(q) = \gamma$ and perform step 1 of the algorithm.
2. When propagating a new binding for a procedure formal that is invoked at a dynamically bound call, we may encounter in the actual parameter list some procedure variable *pv* that is not a formal. In such instances, we assume $Boundto(pv) = \gamma$ and execute step 2b accordingly.

A call invoking a procedure variable generates $|\gamma| = P$ edges in the graph. However, because E represents the edges in the final graph, the time complexity for the propagation algorithm is still $O(N + EP)$. We have added an additional initialization cost to locate elements of γ and to examine assignments to procedure variables, but these can be accomplished in a single pass over each procedure in the program.

9.2 Further Extensions for Increased Precision

This simple algorithm extension differs from the rest of the paper in that we add elements to γ from every procedure, without proving that the procedure is reachable. We also assume that any procedure constant passed as a parameter may eventually be propagated to a variable appearing on the right-hand side of an assignment. Both these assumptions allow the changes to the original algorithm to be small but also introduce imprecision into the call graph.

We can improve precision while maintaining the same time complexity by constructing γ during propagation. This involves propagating new values for γ through previously located calls and tracking values of procedure formals that appear on the right-hand side of assignment statements.

Much greater precision is possible if we maintain separate γ sets for each procedure variable [Wei80]. Weihl's approach could be made even more precise by analyzing control flow within procedures, using something similar to constant propagation (but with $\cup$ as the meet function rather than $\cap$) [Kil73]. However, separate γ sets mean that we must propagate an additional $O(V)$ global procedure variables on *Worklist* and track newly propagated bindings through the assignments in the procedure. Examining control flow within the procedure will prevent initialization from being performed in a single pass over each procedure. These changes greatly increase the costs of the algorithm.

10 Comparison to Other Work

This section reviews the literature on call graph analysis. We consider several algorithms that compute essentially the same call graph, but less efficiently. We also compare against the costs of more precise solutions.

10.1 Iterative Algorithms

Several algorithms have been proposed that iteratively evaluate a set of simultaneous equations (similar to those in Section 3). When a new binding is located for an invoked procedure variable, an edge is added to the graph. The entire graph is reanalyzed in case bindings along this edge affected the solution elsewhere. The worst case number of times for performing analysis is $O(PE_p)$, where E_p is the number of dynamically bound call sites appearing in the program. This bound represents all possible bindings for all dynamically bound call sites. For each of the following algorithms, the time complexity for a single stage of analysis must be multiplied by $O(PE_p)$ to derive the complexity of the algorithm.

Walter describes the call graph using boolean relations on the procedures, performing transitive closure and composition of the relations on each cycle [Wal76]. Weihl extends Walter's method to handle aliasing, assignment and pointer manipulation [Wei80]. Spillman presents a bit matrix formulation to analyze PL/I that is at least $O(N + E)$ for each cycle [Spi71]. Burke presents an interval analysis formulation [Bur87]. One cycle of interval analysis and updates requires $O(NE_p + dE)$ steps [Hal91]. Shivers describes his approach by defining simultaneous set equations, without presenting an algorithm [Shi88].

10.2 Precise Algorithms

Ryder describes an algorithm to propagate simultaneous bindings of values to procedure parameters [Ryd79]. The algorithm computes a call graph that is precise, assuming all call sites are invoked. It makes only a single forward pass over the program, but it cannot analyze languages permitting recursion. We converted Ryder's algorithm to analyze recursive programs using a work-list approach similar to the one presented here [CCHK90]. Shivers also presents more precise versions of his algorithm, but he retains a bounded amount of path information and propagates the information that is unique to each path [Shi91b]. Retaining these paths increases the cost of the algorithm and bounding the length of the paths reduces the precision.

The time complexity of the precise algorithm is bounded by the number of simultaneous bindings of procedure formals across all procedures in the program. If c_p is the maximum number of procedure formals for any procedure, the algorithm is bounded by $O(N + EP^{c_p})$ time. Even with the assumption that c_p is a constant (used in the previous analyses), the algorithm is still polynomial in the size of the graph. Depending on language features, P^{c_p} could be too large for efficient execution of the algorithm. In particular, for a program that has been converted to CPS form, a parameter is added for the continuation so c_p is increased by 1; P is increased to include

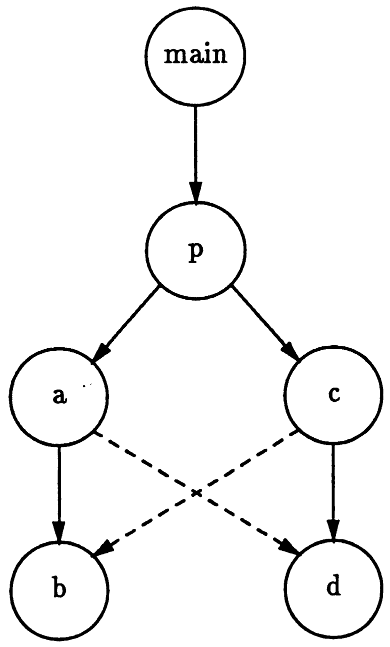


Figure 4 Call graph generated by precise method, with additional edges added by new method shown in dashed lines.

all the continuations in the program.

The difference in the precision of the algorithms is illustrated by the following example.

```
program main          procedure p(e, f)     procedure a(g)     procedure c(h)
    call p(a, b)          call e(f)             call g             call h
    call p(c, d)
```

The call graph for this example is shown in Figure 4. The solid lines show edges added to the graph by either method, and the dashed lines represent additional edges added by the new method.

The extra edges are added because the new algorithm tracks the bindings for the formals *individually*, while the precise algorithm maintains *simultaneous* bindings contributed from each call. Thus, the new algorithm produces a less precise graph whenever there exist two distinct calls to a procedure propagating bindings where at least two of the bindings for the formals differ.[2]

11 Conclusion

This paper has presented a very efficient algorithm for calculating the call graph in the presence of procedure-valued parameters, with a simple extension to handle assignment to procedure variables. The algorithm is efficient because it is demand-driven: as they become available, new values for procedure variables are propagated only to the points that they affect. The algorithm can be used to analyze a broad range of languages. We have implemented the algorithm as part of an

[2]The precise algorithm can be thought of as a specialization of the efficient one. When at most one procedure formal appears as a parameter at a call site, the algorithms are identical.

interprocedural compilation system for scientific Fortran. We have shown that it can also replace the Shivers algorithm in analyzing functional languages such as Scheme, following CPS-conversion.

The new algorithm is less precise than the modified Ryder algorithm. We believe the execution-time cost of the precise algorithm may be too high to merit the gains in precision, depending on language features. For a particular language, these differences will need to be explored. In the case of Fortran, the two algorithms almost always yield the same call graph.

We have not addressed a number of language features affecting call graph analysis. Pointer manipulation of procedure variables requires analysis similar to assignments. This issue has been addressed by other researchers [Wei80, LMSS91]. Aliasing may arise from features other than pointer manipulation. Aliasing due to call-by-reference parameter passing has been widely discussed in the literature. Spillman and Weihl consider its effect on call graph analysis [Spi71, Wei80]. Compiling object-oriented languages contributes more difficult problems to constructing the call graph. Inheritance and function overloading make it difficult to understand what procedures are being invoked even when the function name appears at the call. This problem is addressed by *customization* in SELF in cases where the compiler can statically determine the unique binding of a call site [CU89]. Support for the above language features will increase the cost of the algorithm.

Acknowledgments. The authors wish to thank Tom Murtagh and Rene Rodriguez for their significant contributions to this work.

References

[BCHT90] P. Briggs, K.D. Cooper, M.W. Hall, and L. Torczon. Goal-directed interprocedural optimization. Technical Report TR90-148, Dept. of Computer Science, Rice University, November 1990.

[Bur87] M. Burke. An interval-based approach to exhaustive and incremental interprocedural analysis. Research Report RC 12702, IBM Yorktown Heights, September 1987.

[CCH+88] C.D. Callahan, K.D. Cooper, R.T. Hood, K. Kennedy, and L. Torczon. ParaScope: a parallel programming environment. *International Journal of Supercomputer Applications*, 2(4):84–89, December 1988.

[CCHK90] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, SE-16(4):483–487, April 1990.

[CK88] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pages 57–66. ACM, July 1988.

[CK89] K.D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual Symposium on Principles of Programming Languages.* ACM, January 1989.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 24(7), pages 146–160. ACM, July 1989.

[EB91] R. Eigenmann and W. Blume. An effectiveness study of parallelizing compiler techniques. In *Proceedings of the 1991 International Conference on Parallel Processing, vol. II.* CRC Press, August 1991.

[Fis72] M.J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs,* SIGPLAN Notices 7(1), pages 104–109, 1972.

[Hal91] M. W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, Department of Computer Science, Houston, TX, April 1991.

[HHKT91] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. Technical Report TR91-169, Rice University, Department of Computer Science, November 1991.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems,* 2(3):350–360, July 1991.

[HKM91] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91.* IEEE Computer Society Press, November 1991.

[Kil73] G. Kildall. A unified approach to global program optimization. In *Conference Record of the Symposium on Principles of Programming Languages.* ACM, January 1973.

[LMSS91] J. Loeliger, R. Metzger, M. Seligman, and S. Stroud. Pointer target tracking: an empirical study. In *Proceedings of Supercomputing '91.* IEEE Computer Society Press, November 1991.

[LY88] Z. Li and P. Yew. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems,* SIGPLAN Notices 23(9), pages 85–99, September 1988.

[McK92] K. S. McKinley. *Automatic and Interactive Parallelization.* PhD thesis, Rice University, Department of Computer Science, April 1992.

[Rey72] J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference,* 1972.

[Ryd79] B. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering,* SE-5(3):216–225, May 1979.

[Shi88] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* SIGPLAN Notices 23(7), pages 164–174. ACM, July 1988.

[Shi91a] O. Shivers. *Control-Flow Analysis of higher-order languages.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, May 1991.

[Shi91b] O. Shivers. The semantics of scheme control flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation,* SIGPLAN Notices 26(9), pages 190–198, September 1991.

[Spi71] T. C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *Proceedings of the IFIP Congress 1971,* pages 376–381, Amsterdam, 1971. North Holland.

[TIF86] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* SIGPLAN Notices 21(7), pages 176–185. ACM, July 1986.

[Wal76] K. Walter. Recursion analysis for compiler optimization. *Communications of the ACM,* 19(9):514–516, September 1976.

[Wei80] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh Symposium on Principles of Programming Languages.* ACM, January 1980.