Efficient Call Graph Analysis

Mary Hall
Ken Kennedy

CRPC-TR92223
July 1992

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Appeared in ACM Letters on Programming Languages and
Systems, 1(3), September 1992.

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

Abstract

We present 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.

We discuss the algorithm in relationship to other call graph analysis approaches. Many less efficient techniques produce essentially the same call graph. A few algorithms are more precise, but they may be prohibitively expensive depending on language features.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language constructs and features – *procedures, functions and subroutines*, D.3.4 [**Programming Languages**]: Processors – *compilers*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: *interprocedural data-flow analysis*

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 [7, 11, 12, 14, 22] and compiling for distributed-memory multiprocessors [10]. 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 [19].

*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.

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 representation of 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. The call graph is *actually a multigraph* since each call site 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 compiler constructs the call graph 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 refer to this problem as *call graph analysis*. Dynamically bound calls result from a number of language features including the parameter passing mechanism and variable assignment. 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 [1, 18, 21, 23, 24]. More precise algorithms exist, but they may be prohibitively expensive depending on the language features [3, 17, 20]. While most of these approaches were designed for imperative languages, Shivers used a similar approach to analyze control flow of Scheme in Continuation Passing Style (CPS) form.

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

2 Call Graph Definition

We can determine the structure of the call graph by simulating the parameter passing during execution. We calculate the set *Boundto*($\mathcal{F}$), the procedure constant bindings for each procedure formal $\mathcal{F}$. Thus, *Boundto*($\mathcal{F}$) provides the procedures that may be invoked at call sites to $\mathcal{F}$. Let $\mathcal{C}$ represent a procedure constant, $\mathcal{F}_p$ a procedure formal of procedure p, and c a call site. Then, the *Boundto* sets can be described with the following simultaneous equations.

$$Boundto(\mathcal{C}) = \{\ \mathcal{C}\ \}$$
$$Boundto(\mathcal{F}_p) = \cup_{c\ invokes\ p}\ Boundto(Passed'(\mathcal{F}_p, c))$$

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 call sites invoking the procedure in which it appears.

The function *Passed*$(\mathcal{F}_p, c)$ provides the formal or constant passed to $\mathcal{F}_p$ 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.

$$\textit{Passed'}(\mathcal{F}_p, c) = \begin{cases} p & \text{if } \mathcal{F}_q \text{ is invoked at c} \wedge \textit{Passed}(\mathcal{F}_p, c) = \mathcal{F}_q \\ \textit{Passed}(\mathcal{F}_p, c) & \text{otherwise} \end{cases}$$

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

3 Algorithm

We could easily formulate the above simultaneous equations into a simple iterative algorithm. However, the resulting solution would suffer from the same inefficiency as the other iterative algorithms discussed in Section 9.1. With the location of a new binding for an invoked procedure variable, a new edge is added to the graph. In an iterative approach, this may require reanalysis of the entire program. To avoid this inefficiency, our algorithm propagates new bindings on demand as they are located.

Figure 1 presents the call graph analysis algorithm. The main algorithm *Build* invokes the procedure *InitializeNode* on newly reachable procedures beginning with *main*. For a particular procedure p, *InitializeNode* initializes the *Boundto* sets for procedure parameters of p and locates entries for *Worklist* resulting from statically bound calls in p. *InitializeNode* recursively calls itself to initialize procedures that become reachable through statically bound calls. Thus, the algorithm does not introduce bindings from unreachable call chains.

InitializeNode only locates procedure constant bindings for statically bound calls. The procedure *Build* must propagate bindings for procedure formals at statically bound calls and handle both procedure constants and procedure formals at dynamically bound calls. When *Build* removes a pair $\langle a, \mathcal{F}_p \rangle$ from *Worklist*, it examines all call sites in procedure p. (We assume that procedure p can be recovered from the name $\mathcal{F}_p$.) One of three possibilities exists at call sites referencing $\mathcal{F}_p$.

1. **The call site is statically bound.**
   Then, $\mathcal{F}_p$ 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).

```
/* Add a reachable node and edges for all of its statically bound calls */
procedure InitializeNode(p)
      add p to Nodes
      foreach formal procedure parameter F_p
          Boundto(F_p) ← ∅
      foreach call site c = (p → q) in p
          if q is a procedure constant then
             if q ∉ Nodes then call InitializeNode(q)
             add edge (p → q) to call graph
             foreach procedure constant C ∈ WalkConstants(c)
                foreach F_q in isPassed(C, c, q)
                    add ⟨C, F_q⟩ to Worklist
end /* InitializeNode */


/* Build call graph starting at root procedure */
program Build
     Worklist ← ∅
     Nodes ← ∅
     call InitializeNode(main)

     while Worklist ≠ ∅
          select and delete a pair ⟨a, F_p⟩ from Worklist
          if a ∉ Boundto(F_p) then
             add a to Boundto(F_p)
             foreach call site c = (p → q) in p
(1)              if q ≠ F_p then
                    foreach b ∈ Boundto(q)
                        foreach F_b ∈ isPassed(F_p, c, b)
                            add ⟨a, F_b⟩ to Worklist

(2)              else /* F_p is invoked at the call */
                    if a ∉ Nodes then call InitializeNode(a)
                    add edge (p → a) to call graph
                    foreach procedure-valued actual A ∈ WalkActuals(c)
(2a)                    if (A = F_p) then
                            foreach F_a ∈ isPassed(F_p, c, a)
                                add ⟨a, F_a⟩ to Worklist
(2b)                    else
                            foreach procedure constant C ∈ Boundto(A)
                                foreach F_a ∈ isPassed(A, c, a)
                                    add ⟨C, F_a⟩ to Worklist
end /* Build */
```

Figure 1 Call graph analysis algorithm.

2. **The call site is dynamically bound, but invokes some formal other than $\mathcal{F}_p$.** Again, $\mathcal{F}_p$ 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 $\mathcal{F}_p$.** 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 corresponding parameters of a (step 2b). In the special case that $\mathcal{F}_p$ appears as an actual at the call, we propagate a to the corresponding formal of a (step 2a).

The algorithm relies on a few definitions. A pair $\langle a, \mathcal{F}_p \rangle$ in *Worklist* indicates that a should be added to *Boundto($\mathcal{F}_p$)* and propagated through call sites in p referencing $\mathcal{F}_p$. The function *isPassed* is the inverse of *Passed*. For a procedure formal $\mathcal{F}_p$ that appears in one or more actual parameter positions at call site c in p, *isPassed($\mathcal{F}_p, c, b$)* 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)* is the set of procedure constants passed as actual parameters at call c and *WalkActuals(c)* is the set of procedure-valued actual parameters passed at call c.

4 An Example

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

program *main*	**procedure** $a(f_1, f_2)$	**procedure** $b(f_3, f_4)$
call $a(b, c)$	**call** $b(a, f_2)$	**call** $f_3(f_3, d)$
	call $f_1(f_1, f_2)$	**call** f_4

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 algorithm processes the pairs on *Worklist*. Figure 2 illustrates the execution results, annotating each *Worklist* entry with the step of the algorithm that added it. After the last displayed step, 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.

5 Proofs

In preparation for the correctness and complexity proofs, we define a few terms. E is 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.) N is the number of nodes in the final call graph. P is the number of unique procedure constants passed as parameters in the program. Finally, c_p

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* $\langle c, \mathcal{F}_4 \rangle$: $Boundto(f_4) \leftarrow \{c\}$ 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.

represents the maximum number of procedure-valued parameters for any procedure. We assume c_p is bounded by a small constant, consistent with assumptions made in other work [5, 6].

5.1 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. 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. 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* for all edges 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 $\langle a, \mathcal{F}_{n_m} \rangle$ 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 $\mathcal{F}_{n_m}$ 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 $\langle a, \mathcal{F}_p \rangle$ such that procedure constant a is passed directly to $\mathcal{F}_p$ at some statically bound call site invoking p.

Induction. Assume the lemma is true after the first n iterations of the loop. Then for the pair $\langle a, \mathcal{F}_p \rangle$ selected on iteration $n+1$, a must be an element in the final set *Boundto*($\mathcal{F}_p$). 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 the new binding generates. We visit each call site in p such that $\mathcal{F}_p$ is either invoked or is an actual parameter. The propagation at these calls corresponds to the rules set forth in Section 3. In each case, it can be shown that for any generated *Worklist* pair $\langle b, f \rangle$, $b \in Boundto(f)$ is true. (A detailed proof appears elsewhere [9].)

If $\mathcal{F}_p$ 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, the algorithm inserts pairs $\langle a, \mathcal{F}_{n_m} \rangle$ 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 $\mathcal{F}_{n_m}$, then one binding for $\mathcal{F}_{n_m}$ must be n_{m+1}. By the induction hypothesis, the pair $\langle n_{m+1}, \mathcal{F}_{n_m} \rangle$ must have been added to *Worklist* when the algorithm propagated to n_m actuals passed at c_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 $\langle n_{m+1}, \mathcal{F}_{n_m} \rangle$. 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, eventually will be processed. □

Theorem 1 *Build correctly calculates the Boundto sets as defined in Section 2, 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. □

5.2 Time Complexity

Theorem 2 *The entire call graph analysis algorithm is bounded by $O(N + c_pEP)$ 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. Thus, the initialization step requires $O(c_p(N + E))$.

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 $O(c_pEP)$. Consider the number of unique pairs on *Worklist*. As part of the processing of a pair $\langle a, f\rangle$, the algorithm adds procedure constant a to the *Boundto* set of the procedure parameter f. If this pair is ever selected from *Worklist* again, it will not be processed since a already appears in *Boundto(f)*. Thus, the number of iterations of the *while* loop that proceed past the initial test is no greater than the number of possible unique *Worklist* pairs. These are ⟨ procedure constant, procedure parameter ⟩ pairs, so the bound is $O(c_pNP)$. Since the number of unique *Worklist* pairs for a given procedure is bounded by c_pP, in the *for* loop we visit a distinct call site $O(c_pP)$ times. As a result, examining call sites takes $O(c_pEP)$ time.

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

6 Implementation Results

Our implementation of the algorithm provides the basis for interprocedural optimization in ParaScope, an environment designed for supporting scientific Fortran programmers [2]. 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.

We discovered an important convenience that distinguishes the implementation from the presentation in Figure 1. We added special *representative nodes* to denote calls to procedure formals. When initialization of a procedure locates a call to a procedure formal, we add to the graph a representative node r and an edge from the caller node to r. As a binding for the corresponding procedure formal becomes available, we add an edge from r to the node for the procedure that is potentially invoked at the call. In contrast, the algorithm in Figure 1 would directly add edges from the caller to each of the procedures that may be invoked at the call.

This indirection greatly simplifies accessing interprocedural information about calls through procedure formals. Queries at such a call need only examine information at the edge from the caller to the representative node. Without this indirection, the queries would instead have to form

the meet of the information contributed by all edges corresponding to possible bindings of the procedure formal.

7 Continuation Passing Style and Analyzing Scheme

The previous discussion focuses on languages where procedure variables only receive their values through parameter passing. However, this algorithm is also applicable to languages with procedure-valued returns if the programs are first converted to continuation passing style (CPS).

Continuation passing style (CPS) provides an intermediate program representation where all transfers of control are represented by tail recursive procedure calls [8, 16]. 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. CPS conversion eliminates procedure-valued returns, replacing them with invocations to a formal parameter. This observation is central to Shivers's formulation of control flow analysis of Scheme.

The following example illustrates this point. 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)

This program evaluates the function call *a*(*print*) and passes the result 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, cont)	**procedure** m-c (f)	**procedure** c(f, cont)
call a(print, m-c)	**call** cont(f)	**call** c(f, $\mathcal{K}$)	**call** f(1, cont)

The top continuation $\mathcal{K}$ represents what happens after the program terminates. The added procedure *m-c* represents 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 $\mathcal{K}$ 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.

Once we understood that call graph analysis for Fortran was related to analysis of CPS-converted functional programs, we became interested in the relationship between our algorithm and Shivers's. We discovered two 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 for assignments in the next section. Second, Shivers analyzes incomplete programs where some functions can be invoked by the external environment and invocations appear to external functions. We ignore this issue. 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.

8 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] Assignments to procedure formals are also allowed.

8.1 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 of all the procedures in the program, adding procedure constant $\mathcal{C}$ to γ if $\mathcal{C}$ appears as the right-hand operand of an assignment or if it appears as an actual parameter at a call site[2].

Once γ has been initialized, the algorithm proceeds as before with a few minor changes. There are three changes to *InitializeNode:*

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.
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 formals. When such a procedure formal $\mathcal{F}$ is located, we add a pair $\langle \mathcal{C}, \mathcal{F} \rangle$ to *Worklist* for each procedure constant $\mathcal{C}$ in γ.

There are two changes to *Build:*

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.

---

[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] To improve precision, we could partition γ according to type signatures in type-safe languages or according to number or types of parameters.

8.2 Further Extensions for Increased Precision

This simple algorithm extension differs from the rest of the approach 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 eventually may 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. The details of this change are fairly complex.

Much greater precision is possible if we maintain separate γ sets for each procedure variable [24]. 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 $\cap$) [13]. 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.

9 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.

9.1 Iterative Algorithms

Several algorithms have been proposed that iteratively evaluate a set of simultaneous equations (similar to those in Section 2). 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 affect 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 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 [23]. Weihl extends Walter's method to handle aliasing, assignment and pointer manipulation [24]. Spillman presents a bit matrix formulation to analyze PL/I that is at least $O(N + E)$ for each cycle [21]. Burke presents an interval analysis formulation [1]. One cycle of interval analysis and updates requires $O(NE_p + dE)$ steps [9].

Shivers describes his approach by defining simultaneous set equations [18]. He does not present an algorithm, but the technique was used in his implementation of data-flow analysis in Scheme [19].

9.2 Precise Algorithms

Ryder describes an algorithm to propagate simultaneous bindings of values to procedure parameters [17]. The algorithm computes a call graph that is precise, assuming all call sites are invoked. It cannot analyze languages permitting recursion. In previous work with Carle and Callahan, we converted Ryder's algorithm to analyze recursive programs using a worklist approach similar to the one presented here [3].

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 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. The new algorithm produces a less precise graph whenever: (1) there exist at least two distinct calls to a procedure; (2) the procedure has two or more procedure formal parameters; and (3) each call contributes different bindings for at least two of the formals [3].

In practice, there is little difference between the precise and efficient algorithms when analyzing Fortran. In our experience, Fortran programs almost always contain at most one procedure value that is passed as a parameter (*i.e.*, P $\leq$ 1). The algorithms will perform differently only for languages where procedure variables occur frequently, such as in Scheme.

Shivers also presents precision improvements for his algorithm, but it still may be less precise than the modified Ryder algorithm [20]. He retains a bounded amount of path information and

[3] Using these rules, the compiler could detect potential losses in precision. Imprecision is only possible in procedures with multiple procedure formals where at least two formals have *Boundto* sets containing two or more bindings. The compiler could mark these procedures and any ancestors in the call graph passing two or more procedure-valued actuals along the call chain. By rerunning the efficient algorithm but propagating simultaneous bindings just for calls emanating from the marked nodes, the precise call graph would result. While this change does not improve the time complexity, it could improve the running time of the precise algorithm.

propagates the information that is unique to each path. Retaining these paths increases the space requirements and bounding the length of the paths limits the precision.

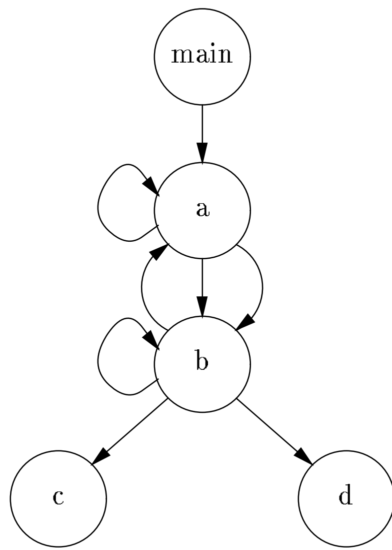
10 Conclusion

We 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 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 [15, 24]. 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 [21, 24]. 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 [4]. Support for the above language features will increase the cost of the algorithm.

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



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

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

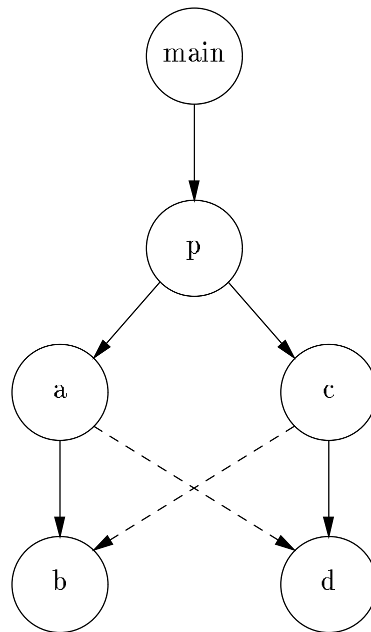


Figure 4 Call graph generated by precise method, with additional edges added by new method shown in dashed lines.

References

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

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

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

[4] Chambers, C. and Ungar, D. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *ACM SIGPLAN Notices*, 24(7):146–160, 1989.

[5] Cooper, K.D. and Kennedy, K. Interprocedural side-effect analysis in linear time. *ACM SIGPLAN Notices*, 23(7):57–66, 1988.

[6] Cooper, K.D. and Kennedy, K. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual Symposium on Principles of Programming Languages*, pages 49–59. ACM, January 1989.

[7] Eigenmann, R. and Blume, W. An effectiveness study of parallelizing compiler techniques. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II–17—II–24. CRC Press, Inc., August 1991.

[8] Fischer, M.J. Lambda calculus schemata. *ACM SIGPLAN Notices*, 7(1):104–109, 1972.

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

[10] Hall, M.W., Hiranandani, S., Kennedy, K., and Tseng, C. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*. IEEE Computer Society, November 1992.

[11] Hall, M.W., Kennedy, K., and McKinley, K.S. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, pages 424–434. IEEE Computer Society, November 1991.

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

[13] Kildall, G. A unified approach to global program optimization. In *Conference Record of the Symposium on Principles of Programming Languages*, pages 194–206. ACM, January 1973.

[14] Li, Z. and Yew, P. Efficient interprocedural analysis for program restructuring for parallel programs. *ACM SIGPLAN Notices*, 23(9):85–99, 1988.

[15] Loeliger, J., Metzger, R., Seligman, M., and Stroud, S. Pointer target tracking: an empirical study. In *Proceedings of Supercomputing '91*, pages 14–23. IEEE Computer Society, November 1991.

[16] Reynolds, J.C. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, 1972.

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

[18] Shivers, O. Control flow analysis in Scheme. *ACM SIGPLAN Notices*, 23(7):164–174, 1988.

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

[20] Shivers, O. The semantics of Scheme control flow analysis. *ACM SIGPLAN Notices*, 26(9):190–198, 1991.

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

[22] Triolet, R., Irigoin, F., and Feautrier, P. Direct parallelization of call statements. *ACM SIGPLAN Notices*, 21(7):176–185, 1986.

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

[24] Weihl, W.E. 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*, pages 83–94. ACM, January 1980.