Constructing call multigraphs using dependence graphs

Arun Lakhotia

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 231-6766, -5791 (Fax)
arun@cacs.usl.edu

1 Introduction

A *call multigraph*[*] of a program is a directed multigraph encoding the possible calling relations between procedures. These graphs are used in interprocedural program optimization [2, 3, 9, 15] and for reverse engineering of software systems [7, 8]. For programs that do not contain procedure valued variables (referred to henceforth as procedure variables) this graph can be constructed by a single pass over the program collecting the procedures called at each call site. When procedure variables and indirect calls using values of such variables are allowed constructing such a graph is not so simple. In the worst case, the value of a procedure variable at a call site may be a reference to any procedure in the program. For interprocedural optimizations and for understanding programs one would like to have more precise solutions.

The importance of precisely constructing an analogue of call graph (referred to as the 0^{th} order control flow analysis or 0CFA) in the context of higher order languages such as Scheme and ML has been eloquently elaborated by Shivers [18]. A precise call graph enables data flow optimizations that in turn enables efficient implementations of programs in these languages. The precision of a program's call graph affects the precision of interprocedural analysis in Fortran compilers as well [5].

This paper describes a new polynomial time algorithm for constructing such a call graph that is precise within the limitations of *flow insensitive* interprocedural analysis. The key aspect of our solution is our model of the problem as a constant propagation problem over the domain - powerset of all procedure constants with set union as the meet operator. We call this the problem of propagating sets of procedure-*values* and show that this problem, unlike constant propagation problem, belongs to the class of distributive flow analysis problems [12]; it is therefore decidable.

Our algorithm performs interprocedural flow analysis [10] to construct the graph. We develop an interprocedural procedure values propagation algorithm by amalgamating Wegman and Zadeck's constant propagation algorithm [20] and Horwitz, Reps, and Binkley's interprocedural forward slicing algorithm [11]. Interprocedural analyses themselves depend on call multigraph. Our algorithm resolves the conflict by iteratively propagating procedure-values over a system dependence graph representation of a program [11] and constructing the call graph, till a fixed point is reached. We give a formal definition of the term *precise* call graph and prove that the call graph constructed by our algorithm is precise. Our algorithm computes precise call graphs for a larger

[*] In this paper call multigraph is also referred to as the call graph.

This work was supported by the grant LEQSF (1991–92) ENH-98 from the Louisiana Board of Regents.

ACM-20th PoPL-1/93-S.C., USA
© 1993 ACM 0-89791-561-5/93/0001/0273...$1.50

```
procedure P(X, Y);
     proc var X, Y;
     X := Y;
end;

procedure Q(),....
procedure R();....
procedure S(Z);
     proc var Z;
I4:      *Z();
end;
```

```
procedure main();
    proc var A, B;
    proc ref P, Q, R, S;
C1:     P(A, Q);
I1:     *A();
C2:     P(B, R);
C3:     S(B);
C4:     P(A, P);
I2:     *A(A, S);
I3:     *A(Q);
end;
```

Figure 1 A program with procedure variables and indirect calls.

```
if A in {p1, p2} then {
     if A == p1 then p1(x1, x2, ...);
     else if A == p2 then p2(x1, x2, ...);
} else {
     if A == p3 then p3(x1, x2, ...);
     else if A == p4 then p4(x1, x2, ...);
     else if A == p5 then p5(x1, x2, ...);
     ...
     else terminate abnormally.
}
```

Figure 2 Code segment used to reify an indirect call $*A(x1, x2, ...)$ at site c_i where $M(c_i) = \{p_1, p_2\}$.

class of programs than the works in [4, 5, 17, 18, 19, 21].

The rest of the paper is organized as follows. Section 2 establishes the terminology and formulates the problem. Section 3 discusses the previous works on this problem and elaborates upon our contribution. Section 4 summarizes Horwitz et. al.'s system dependence graph (SDG) and our extensions. For more details about SDGs the reader is referred to [11]. Section 5 gives our interprocedural procedure-values propagation algorithm. Section 6 gives the algorithm to construct the call graph. Section 7 proves that our algorithm computes precise call graph and analyzes its complexity. It is followed by our conclusions and the list of references.

2 Problem formulation

We consider programs written in a procedural language that permits indirect calls through procedure valued variables. Parameters across a procedure call are assumed to be passed by call by value-result. Global variables are not permitted. Only variables are allowed in the actual parameter list of procedure calls and repetition of variables in this list is not allowed. These restrictions prohibit aliasing in programs. The language, parameter transfer mechanism, and these restrictions are all inherited from Horwitz et. al. [11] since we use their system dependence graph for representing programs. These restrictions can be removed using the methods suggested by them.

The program in Figure 1 will be used as a running example. The statements starting with an * are indirect call statements. Further, it is assumed that a special procedure *main* exists that initiates the execution.

Definition: (Domain constraint). A procedure variable in a program can only be defined using assignment statements that either have a procedure reference or a procedure variable on the right hand side. ♡

This constraint prohibits one to assign numeric expressions that evaluate to the physical memory address of a function as one may do in programming languages such as C.

Definition: (Completeness constraint). Only procedures contained in a program may be called from a call site inside the program. ♡

In the absence of this constraint a procedure may call another procedure *external* to the program which may then call some procedure within the program. Such a call will go undetected.

Definition: Let $P = \{p_1, p_2, ..., p_n\}, n > 0$ be the set of procedures in a program (including *main*). We associate a unique identifier to all statements of a program. Let $C = \{c_1, c_2, ..., c_m\}, m \geq 0$, be the set of (identifiers of the) indirect call vertices in the program. Let $\mathcal{P}$ be the power set of P. ♡

Definition: The problem of constructing a call graph is essentially the same as creating a mapping $M : C \rightarrow \mathcal{P}$, that maps each indirect call site to a set of procedure references constituting the procedures that *may* be called from that site. For c_i this set is denoted by $M(c_i)$. ♡

Definition: A call graph is *dynamically precise* if for every call site c_i, $M(c_i)$ contains only those procedures that can be called from c_i for some initial state σ. ♡

This definition is not useful since it relies on dynamic characteristics of the program. We would like a definition that depends on a program's static characteristics. The obvious definition based on paths in the flowgraph also does not suffice since due to presence of indirect calls the paths are not known statically. We solve the problem by transforming the program P to P_M to P_M^∞ such that P_M is free of indirect procedure calls and P_M^∞ has no procedure calls at all. The three programs are equivalent under declarative semantics.

Note that P_M^∞ is introduced only to define the notion of *statically precise* call graph. It is only manipulated mathematically to establish properties of our algorithm.

Definition: Let P_M denote the program due to expanding indirect call statements in program P with respect to a mapping M as follows. If c_i contains $*A(x1, x2, ...)$ and $M(c_i) = \{p_1, p_2\}$ then the call in c_i is replaced by the code segment in Figure 2. All other statements of P are copied as is to P_M. ♡

Definition: P_M^∞ is the potentially infinite single procedure due to exhaustive inlining of procedure calls inside procedure *main* of program P_M. ♡

The outer *if* introduced by expanding a c_i in P_M does not contribute to its meaning but serves to create different paths for procedures contained in $M(c_i)$ and those not contained in it. The statements in the *then* side of an outer *if* statement in P_M (not P_M^∞) are said to lie on the *true* branch and those on the *else* side in the *false* branch.

Definition: An *M-path* in the flowgraph of P_M^∞ is a path in it that does not contain any statement that is an instance of some statement in the *false* branch of P_M. (Refer to [1] for definitions of flowgraph and path). ♡

Definition: A mapping M is *conservative* iff it has the property:

if $\exists$ a statement s_1 in P_M^∞ such that

a. s_1 contains $v_1 := p_j$,
b. in the flow graph of P_M^∞ there is an M-path from s_1 to c_i' (an instance in P_M^∞ of the outer *if* statement in P_M that corresponds to c_i),
c. a sequence of 'identity' assignments copy the value of s_1 into the variable used in c_i', and
d. there is an $M-$path from *start* vertex of the flowgraph to s_1

then $p_j \in M(c_i)$.

♡

The condition says that if the program execution takes a path such that some statement s_1 propagates procedure reference p_j to c_i' then the call graph should state that p_j may be called from c_i. But reification of indirect call sites may introduce paths in the program that may not really happen. The condition "there is an M-path from the *start* vertex to s_1" removes such paths from consideration.

When the definition is applied from procedure *main* onwards, we claim that it will require all procedures that are called from a call site to be included in the call graph. The mapping is conservative since $M(c_j)$ *may* contain procedures whose values can not be propagated to c_j through any path starting from *main*.

Definition: A mapping M is *optimistic* iff it satisfies the property resulting from swapping the antecedent and the consequent of the definition of conservative. ♡

This is the inverse of the definition for conservative. It says that $M(c_i)$ should not contain a procedure reference if there is no path through which its value can be propagated to c_i.

Definition: A mapping is *statically precise* if it is both conservative and optimistic.

We show that the call graph computed by our algorithm is *precise* in the above sense.

3 Comparison with previous works

The call multigraph construction problem has previously been studied by Ryder [17], Burke [4][‡], and Callahan et. al. [5], Spillman [19] and Weihl [21]. Our work may be compared with these on a) constraints imposed on the usage of procedure variables and b) the precision of the resulting call multigraph.

The work of Burke, Ryder, and Callahan are specific to Fortran in which a) procedure valued variables are only allowed as procedure parameters and b) assignments to formal procedure parameters is not allowed, they can only receive procedure values from actual parameters. Amongst them, Burke's and Callahan et. al.'s algorithms work for recursive Fortran programs where as Ryder's algorithm does not. Further, Callahan et. al.'s and Ryder's algorithms return more precise call graphs than Burke's.

The program in Figure 1 uses procedure variables in a way not permissible in Fortran, hence Burke, Callahan et. al., and Ryder's [4, 5, 17] algorithms can not construct its call graph. Our algorithm is therefore applicable for a larger class of languages than these works. If, however, our algorithm is applied to the same class of language as Callahan et. al.'s and Ryder's work, respectively, it is our *conjecture* that results returned will identical.

Spillman [19] and Weihl [21] work with a procedural language that permits label valued variables and aliasing. Due to the presence of label variables they do not even have the flow graph of a program. As a result they can not assume any ordering in a program's statements. The results of the analysis are very imprecise. The call graph constructed by our algorithm and that due to Weihl's are given in Table 1 (towards the end of the paper). The reader may notice the difference in the results.

‡ The chapter concerning construction of call multigraph in [4] has been omitted from its journal version published in ACM TOPLAS (July, 1990).

4 System dependence graph

Horwitz et. al.'s SDG encodes the data, control, and call dependence relations between statements[§] of a program in a simple procedural language stated in Section 2 [11]. The next paragraph outlines the various types of vertices and edges in an SDG as defined by Horwitz et. al. For a more detailed description the reader is referred to [11].

The SDG consists of a collection of procedure dependence graph (PDG) (a variation of program dependence graph [14, 16]). There is one PDG per procedure in the program encoding the control and data dependence relations within the procedure. The various types of vertices in a PDG are: for statements – *assignment*, *if*, *while*, *finaluse*, for procedure call – *call site*, *actual-in*, *actual-out*, and for procedure entry - *entry*, *formal-in*, *formal-out*. The edges connecting vertices within a PDG are *control* and *flow* edges. The edges between vertices of different PDGs are: *call edge* – from a *call-site* to an *entry* vertex; *parameter-in* edge – from an *actual-in* to an *formal-in* vertex; *parameter-out* edge – from *formal-out* vertex to *actual-out* vertex; and *summary edge* from an *actual-in* to an *actual-out* vertex.

Our extension. We extend Horwitz et. al.'s definition of SDG to contain *indirect call* vertices to represent an indirect call. Each indirect call vertex has its pairs of actual-in and actual-out vertices that have flow edges connecting them to the vertices within the PDG. Since the procedures called from an indirect call statement are not known its vertices are not connected to vertices of any procedure entry and there are no summary edges between the actual-in and actual-out vertices.

Our algorithm to construct call graph is iterative. As procedures that may be called from an indirect call site are detected the SDG is modified to represent this knowledge. For each pro-

§ It also has another dependence called the def-order dependence which is not relevant for our work. This dependence is therefore ignored in this paper.

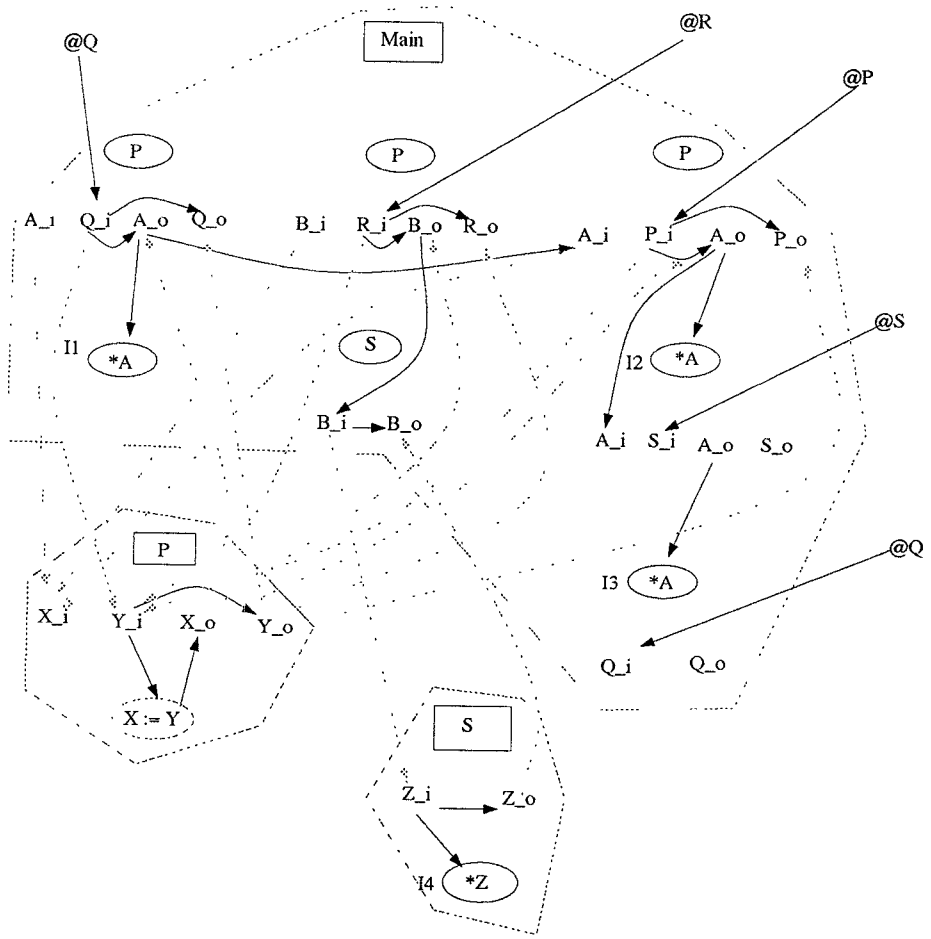

Figure 3 SDG of program in Figure 1.
Legend for vertices: rectangular boxes - *entry*; ovals - direct and indirect calls; dotted oval - other statements; @X - not a true vertex, shows usage of reference to procedure X; A_i or A_o - parameter vertex (actual or formal depending on geometric proximity with entry or call vertex), *_i* denotes *_in* and *_o* denotes *_out*; dotted polygon - PDG of each procedure. Legend for edges: shaded arrows - parameter-in and parameter-out edges; solid arrows - data dependence edges. To avoid clutter the control, indirect-control, and call edges are not shown.

cedure that may be called from an indirect call site we introduce a *virtual call-site* and connect it by an *indirect control edge* emanating from the corresponding indirect call site. The virtual call-sites too have actual-in and actual-out vertices and they may be connected by summary edges. Flow dependence edges entering or exiting the actual parameter vertices of a virtual call site are copied from those of the indirect call site. These vertices are also connected by parameter-in and parameter-out edges to the formal vertices of the corresponding procedure.

It may be reemphasized that the virtual call site vertices are created by our algorithm. In the initial SDG only (direct) call sites are connected to the entry vertices of the corresponding procedures. This SDG can be created using Horwitz et. al.'s algorithm. Figure 3 gives the initial SDG of the program in Figure 1.

5 Propagating procedure values using SDG

The goal of constant propagation is to "discover values that are constant on all possible executions of a program and to propagate these constants as far as through the program as possible" [6]. The constant propagation algorithms of [6, 13, 20] use a *flat* lattice and the following meet rules:

```
global G /* an SDG */;
procedure PropagateProcedureValues; {
    /* Put 'root' nodes in Worklist */
    Worklist := φ
    put nodes in G containing p:=c into Worklist, where p is a procedure variable and c∈ P;
    unmark all nodes
    /* Propagate constants to calling procedures */
    propagateUsingSelectiveEdges(Worklist, [flow, summary, parameter-in]);
    let Worklist be the set of all marked nodes;
    /* Propagate constants to called procedures */
    propagateUsingSelectiveEdges(Worklist, [flow, summary, parameter-out]);
}
procedure propagateUsingSelectiveEdges(Worklist, EdgeTypes); {
/* propagate constants from vertices in Worklist by traversing only edges of type in EdgeTypes */
    while Worklist ≠ φ do {
        select and remove an element v from Worklist
        for every vertex w such that v → w is an edge whose type is one of EdgeTypes do {
            if w.Value ≠ w.Value ∪ v.Value
                w.Value := w.Value ∪ v.Value;
                put w in Worklist;
        }
    }
}
/* Initializations to be done before PropagateProcedureValues is called */
procedure Initialize; {
    for every node s in G do {
        if s is an indirect call site: s.Value := φ;
        if s assigns a procedure variable to a procedure variable: s.Value := φ;
        if s assigns a procedure reference c to a procedure variable: s.Value := c
    }
}
```

Figure 4 Algorithm for propagating procedure values using SDG.

$$a \wedge b = \top \text{ if } a \neq b$$
$$a \wedge a = a$$
$$a \wedge \top = \top$$
$$a \wedge \bot = a$$

If the *flat* lattice is replaced by the *power set of all values* in a given domain and the meet operator by *set union* these algorithms may not terminate for infinite domains such as integers or real numbers. However, if the domain is finite (as is the set of all procedures in a given program) the algorithms will terminate [12] and return the *set of all values* a variable at a given site in the program may take. To differentiate with the *constant* propagation problem we call this the problem of propagating *procedure values*.

The domain constraint of Section 2 stated in Kam and Ullman's [12] terminology implies that the function space for this problem consists of only the identity function. It is therefore a *distributive* flow analysis problem and, unlike traditional constant propagation, can be computed precisely. The reader may note that since the call multigraph problem is formulated in terms of the flow graph of P_M^{∞} its computability may be analysed using Kam and Ullman's framework. Our

```
procedure ConstructCallGraph; {
    construct PDGs for all procedures
    Initialize; (See Figure 4)
    repeat
        Construct the SDG using Value of procedure variables at indirect call-sites
        PropagateProcedureConstants (see Figure 4)
    until Value of no indirect call-site changes
    /* create call graph M) is given */
    ∀c_t ∈ C : M(c_t) = c_t.Value.
}
```

Figure 5 Algorithm for constructing call graph

algorithm however does not use flow graph.

Figure 4 gives an algorithm to propagate *sets of values* of domain $\mathcal{P}$ for our extension of Horwitz et. al.'s system dependence graph [11]. The procedure *Initialize* should be called before *PropagateProcedureValues* is called. It has been kept outside because in the next section *PropagateProcedureValues* is called in a loop where the initializations need only be performed once. Examples enumerating the working of the algorithm are also presented in the next section.

Horwitz et. al. have noted that indiscriminate traversal of an SDG's edges can create a dependence path between vertices of two procedures even when none exists. This happens because a procedure entry vertex may be connected to multiple call sites. A traversal may use parameter-in edge coming out of one call site and parameter-out vertex returning to another call site to create the incorrect linkage. Horwitz et. al. termed this the *calling context problem* and developed a two pass traversal to solve it for their interprocedural *forward* slicing algorithm [11]. The first pass of their algorithm uses only control, flow, summary, and parameter-out edges for traversing the graph. Similarly, the second pass uses only control, flow, summary, call, and parameter-in edges. Since we need only data dependence we drop the control and call edges from the two passes; which explains the choice of parameters to *propagateUsingSelectiveEdges* in Figure 4.

The *Worklist* to maintain vertices to be processed and the condition when vertices may be added to it is retained from Wegman and Zadeck's [20] algorithm. Due to the domain constraint, an assignment statement (of interest to us) can only have one variable on the right hand side. This removes the need to introduce *join nodes*, as done by Wegman and Zadeck.

We have not used Callahan et. al.'s interprocedural constant propagation algorithm [6] for our problem because it gives results less precise than our algorithm. This is because Callahan et. al.'s summary information is computed by ignoring the existence of other call sites. Our algorithm has access to the PDG of the procedure and hence there is no approximation. Our algorithm can however not be used to propagate constants when the domain constraint is removed.

6 Constructing call graph

The algorithm for finding the set of procedures called from an indirect call-site is given in Figure 5. The algorithm is iterative. The *Value* field of an indirect call-site gives the set of procedures known to be called from that site. Before the first iteration this is initialized to the empty set. The SDG is updated at each iteration to add virtual call sites corresponding to the *Value* of the indirect call sites. The procedure values are then propagated over the SDG potentially changing the *Value* of some indirect call site. The SDG

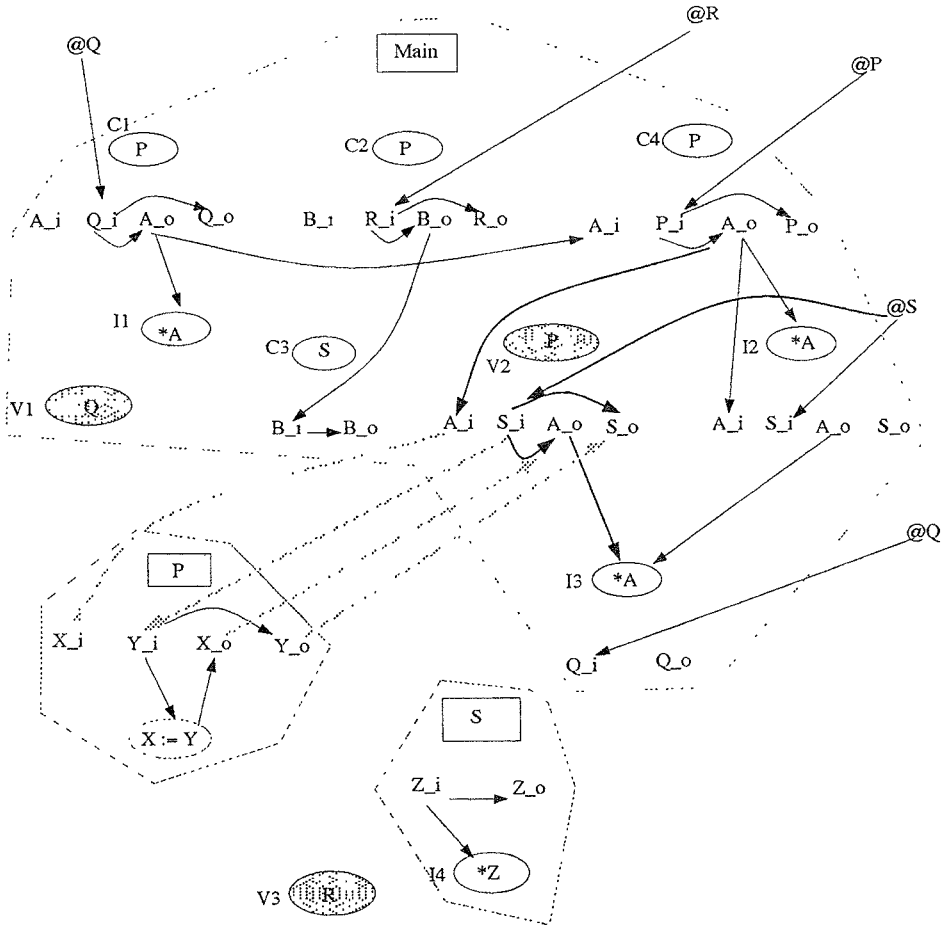

Figure 6 *Partial* SDG after first iteration.
Legend: shaded ovals - indirect call-sites; thicker solid arrows - new flow edges added after this iteration. shaded arrows - new parameter in/out edges; Not shown: edge from indirect call-site to virtual call-site. Adding nodes and edges from previous iteration will make this SDG complete.

is updated again and the process iterated until the *Value* of all the indirect call-sites stabilize.

We now apply the algorithm to the program in Figure 1 whose initial SDG is given in Figure 3. In this SDG the PDGs for procedures Q and R and all the control and def-order edges have been ignored. The nodes of the indirect call sites have been labelled *I1*, *I2*, *I3*, and *I4*. The program requires four iterations of the *repeat* loop to propagate all the procedure constants. Figures 6 to 8 give the new SDG due to *Value* after the end of each iteration. The virtual call-sites created as more *Value* of indirect call-sites get known are shown in shaded ovals. They are labelled *V1*, *V2*, *V3*, *V4*, and *V5*. Table 1 gives the *Value* for these nodes initially and after each iteration. For the sake of comparison, it also contains the values generated using Weihl's algorithm [21]. The other call graph construction algorithms do not process programs in our language and hence their results for this program can not be compared against ours.

At the end of the first iteration, the procedure references Q, R, and P from call statements *C1*, *C2*, and *C4* are propagated to the indirect call-sites *I1*, *I4*, and *I2*, respectively (see Figure 6). Notice that *I4* an indirect site in procedure *S* is connected to call site *C2* of *Main* through another call site *C3* (also in *Main*). Similar propagation of constants across procedures through call sites in the same procedure is not permissible by Callahan et. al.'s interprocedural constant propagation algorithm [6]. Virtual call sites *V1*, *V2*, and *V3* are introduced in the SDG to reflect the

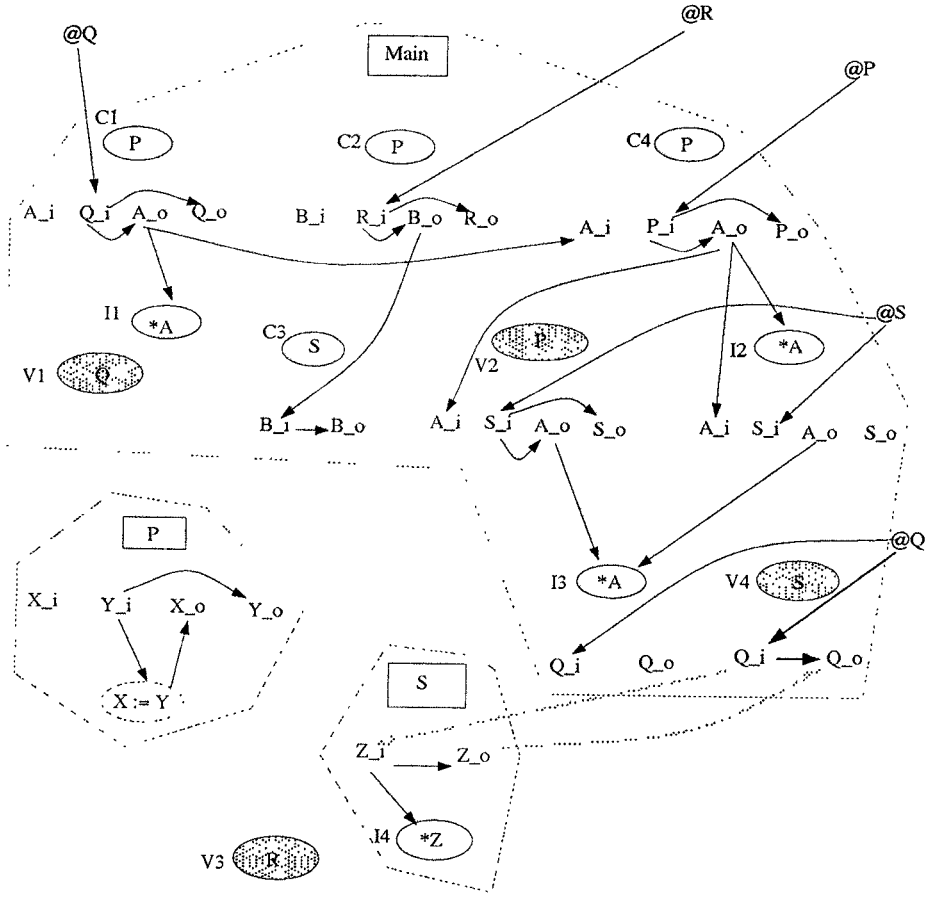

Figure 7 Partial SDG after second iteration

new knowledge.

The second iteration (see Figure 7) propagates procedure reference *S* from the virtual call-site *V2* (related to *I2*) to another indirect call-site *I3*. A new virtual call-site *V4* is introduced.

The third iteration propagates a procedure reference *Q* from the virtual call-site *V4* (related to *I3*) to indirect call-site *I4*. A new virtual call site *V5* is introduced. In the fourth iteration no more values are propagated. Hence there is no change in the SDG and the algorithm terminates.

7 Correctness and complexity analysis

The algorithm of Figure 5 is guaranteed to terminate because a) the program is assumed to be complete, b) there are finite number of procedures, and c) the maximum number of times the value of a vertex can change is $\mid P \mid$ (the maximum length in the lattice $(\mathcal{P}, \subseteq)$).

We now prove that the call graph created by our algorithm is statically precise. The proof is based on two conjectures a) the SDG adequately represents a program's dependencies and b) the SDG traversal algorithm only traverses data dependence paths. The interprocedural slicing algorithm of Horwitz et. al. [11] is based on these conjectures.

Our algorithm may be viewed as creating a sequence of call graphs $M_0, M_1, ..., M_\pi$ where

$$M_i(c_j) = c_j.Value, \forall c_j \in \mathcal{C}$$

before the i^{th}iteration.

The gist of the proof is as follows. M_0 is optimistic because $\forall c_j, M_0(c_j) = \phi$. We prove that if M_i is optimistic then so is M_{i+1}. Finally, M_π, the call graph on termination of the algorithm is conservative because there are no procedure

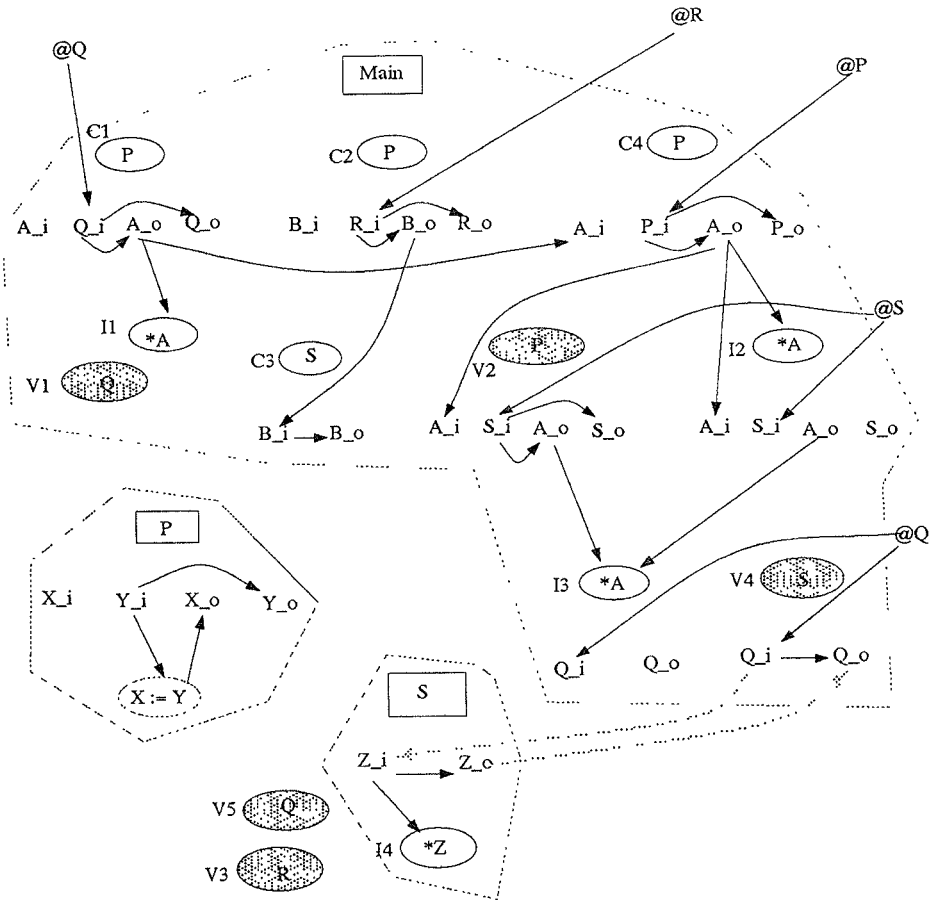

Figure 8 Partial SDG after third iteration

values that may reach an indirect call site c_j and are not already in $M(c_j)$. Hence M is precise.

Since the precise call graph is defined in terms of P_M^∞ and M-paths, while our algorithm uses system dependence graph, we first draw a relation between the two. This requires introducing some more notations.

Definition: $\tau(P_M)$ is the program created by removing from P_M the **else** part of all the **if** statements introduced after replacing the indirect call sites.

Notice the relation between P_M^∞ and $\tau(P_M)^\infty$. All the paths in $\tau(P_M)^\infty$ are M-paths and all the M-paths of P_M^∞ are also contained in $\tau(P_M)^\infty$.

Definition: Let G_M denote the SDG for P with virtual call sites for the call multigraph defined by M.

Definition: Lett G_M^∞ be the potentially infinite graph created by replacing a *call-site* and the *actual* parameter vertices by the corresponding *entry* and *formal-parameter* vertices.

Conjecture 1: G_M^∞ is isomorphic to the PDG of $\tau(P_M)^\infty$.

From this conjecture one may say that if in P_M^∞ the value of a variable defined at a statement x reaches a statement y then there will be a data dependence edge in G_M^∞ from vertex corresponding to x to that corresponding to y.

Conjecture 2: When traversing an SDG using Horwitz et. al.'s interprocedural forward slicing algorithm, if a vertex v is found to be in the slice of a vertex w then in G_M^∞ there exist vertices v' and w', instances of vertices v and w, respectively, such that there is a path from v' to w'.

Since we only use a subset of the type of edges used in each pass of Horwitz et. al.'s forward slicing algorithm it can be inferred that if there is a path in our system dependence graph

that path is also traversed by Horwitz et. al.'s algorithm (with appropriate slicing criterion).

From the above conjectures we can infer the following. Every path (sequence of edges) of G_M traversed by our algorithm corresponds to a set of paths in G_M^∞ which only represent flow of data along some paths in $\tau(P_M)^\infty$ and therefore along M-paths of P_M. By definition, in the flow graph corresponding to $\tau(P_M)^\infty$ there is a path from the start vertex to all the vertices. Hence, every vertex of G_M used in our algorithm to create the initial $Worklist$ is reachable from the start vertex over an M-path in the corresponding flowgraph.

Lemma: *If M_i is optimistic then M_{i+1} is optimistic.*

Proof: For all $c_k \in C$ if there exists a procedural value p_j in $M_i(c_k)$ then it also exists in $M_{i+1}(c_k)$. If the four conditions for the optimality of M are true in $P_{M_i}^\infty$ then they are also true in $P_{M_{i+1}}^\infty$.

We therefore only need to concern about the condition when there exists p_j in $M_{i+1}(c_k)$ and not in $M_i(c_k)$.

For a procedure value to be added to the *Value* field of any vertex in the $i+1^{th}$ iteration there must exist an assignment statement that assigns the value to a variable and there must exist a path in $G_{M_i}^\infty$ along which the value reaches that vertex. These paths also exist in $G_{M_{i+1}}^\infty$. Since all paths in a G_M^∞ are M-paths and since the domain constraint allows only identity assignments to procedure variables the four conditions in the definition of optimistic constraint are satisfied. Hence, if M_i is optimistic then so is M_{i+1}. ♡

Theorem: *The call graph $M = M_\pi$ created by algorithm of Figure 5 is statically precise.*

Proof: Follow from above Lemma and that termination implies M_π is conservative. ♡

Complexity analysis. Let there be N vertices (including parameter vertices) that define procedure variables. Since maximum times the value of a vertex can be changed is $\mid P \mid$, the SDG may at worst be created $N \cdot \mid P \mid$ times. The computation of SDG can be done in polynomial time [11] hence the construction of a call graph may also be done in polynomial time.

Table 1 *Values* of indirect-call sites initially and at the end of each iteration. The last column shows the mapping due to Weihl's method.

Call-site	Initially	Iteration 1	Iteration 2	Iteration 3 & 4	Weihl's method
I1	ϕ	Q	Q	Q	P,Q,R,S
I2	ϕ	P	P	P	P,Q,R,S
I3	ϕ	ϕ	S	S	P,Q,R,S
I4	ϕ	R	R	R,Q	R,Q

Notice, that in each iteration new vertices and edges are added to the SDG, they are never removed. Intermediate information extracted by the SDG construction algorithm can be saved so that successive iterations may only perform incremental work to *update* the SDG. The complexity of the resulting algorithm then adds (not multiplies) a polynomial to the worst case complexity of Horwitz et. al.'s algorithm. More details are beyond the scope of this paper.

8 Conclusions

We revisit the problem of constructing call multigraph of a program that contains indirect calls to procedures using procedure variables. The problem has previously been looked at by Spillman [19], Ryder [17], Weihl [21], Burke [4], and Callahan et. al. [5]. Shivers [18] solves an equivalent problem in the domain of functional languages and terms it zeroth order control flow analysis (0CFA).

We present a polynomial time algorithm that gives statically precise call multigraph for a larger class of procedural language than those dealt by Ryder, Burke, Callahan et. al., Spillman, and Weihl's. Our algorithm is significant because the precision of interprocedural data flow analyses performed by optimizing compilers depend on the precision of the call graphs they compute. Call graphs are also used for reverse engineering of software systems and provide cross reference for

program understanding. The precision in computing such a graph can also significantly help these activities.

Acknowledgments: We thank Rajiv Bagai and William Landi on their comments on an earlier version of this paper. The referees comments were helpful in improving the presentation of this paper.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Allen, F. E. Interprocedural data flow analysis. In *Proceedings IFIP Congress, 1974* (1974), North-Holland, pp. 398–402.

[3] Banning, J. P. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of 6th Annual Symposium on Principles of Programming Languages* (1979), ACM, pp. 29–41.

[4] Burke, M. An interval-based approach to exhaustive and incremental interprocedural analysis. Tech. Rep. RC 12702, IBM Research Center, Yorktown Heights, NY, Sept. 1987.

[5] Callahan, D., Carle, A., Hall, M. W., and Kennedy, K. Constructing the procedure call multigraph. *IEEE Trans. Softw. Eng. 16*, 4 (Apr. 1990), 483–487.

[6] Callahan, D., Cooper, K. D., Kennedy, K., and Torczon, L. Interprocedural constant propagation. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction* (June 1986), pp. 152–161.

[7] Chikofsky, E. J., and Cross II, J. H. Reverse engineering and design recovery: A taxonomy. *IEEE Software* (Jan. 1990), 13–17.

[8] Choi, S. C., and Scacchi, W. Extracting and restructuring the design of large systems. *IEEE Software* (Jan. 1990), 66–71.

[9] Cooper, K. D., and Kennedy, K. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction* (June 1984), pp. 247–258.

[10] Hecht, M. S. *Flow Analysis of Computer Programs.* North-Holland, New York, 1977.

[11] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12*, 1 (1990), 26–60.

[12] Kam, J. B., and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7* (1977), 305–317.

[13] Kildall, G. A unified approach to global program optimization. In *Proceedings of the first ACM Symposium on Principle of Programming Languages* (Oct. 1973), pp. 194–206.

[14] Kuck, D. J., Muraoka, Y., and Chen, S. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Transactions on Computers C-12*, 12 (Dec. 1972).

[15] Myers, E. A precise interprocedural data flow algorithm. In *Proceedings of the 8th Annual Symposium on Principles of Programming Languages* (Jan. 1981), pp. 219–230.

[16] Ottenstein, K. J., and Ottenstein, L. M. The program dependence graph in a software development environment. *ACM SIGPLAN Notices 19*, 5 (May 1984).

[17] Ryder, B. G. Constructing the call graph of a program. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 216–226.

[18] Shivers, O. *Control-flow Analysis of Higher-Order Languages.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[19] Spillman, T. C. Exposing side-effets in a PL/I optimizing compiler. In *Proceedings IFIPS (Computer Software) Conference* (1971), pp. 56–60.

[20] Wegman, M., and Zadeck, F. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13*, 2 (Apr. 1990), 181–210.

[21] Weihl, W. E. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages* (Jan. 1980), pp. 83–94.