Demand-Driven Construction of Call Graphs*

Gagan Agrawal

Department of Computer and Information Sciences, University of Delaware
Newark DE 19716
(302)-831-2783
agrawal@cis.udel.edu

Abstract. Call graph construction has been an important area of research within the compilers and programming languages community. However, all existing techniques focus on exhaustive analysis of all the call-sites in the program. With increasing importance of just-in-time or dynamic compilation and use of program analysis as part of the software development environments, we believe that there is a need for techniques for demand-driven construction of the call graph. We present a demand-driven call graph construction framework in this paper, focusing on the dynamic calls due to polymorphism in object-oriented languages. We use a variant of Callahan's Program Summary Graph (PSG) and perform analysis over a set of influencing nodes. We show that our demand-driven technique has the same accuracy as the corresponding exhaustive technique. The reduction in the graph construction time depends upon the ratio of the cardinality of the set of influencing nodes to the set of all nodes.

1 Introduction

With increasing focus on interprocedural optimizations in compilers for imperative and object-oriented languages and with increasing use of results of interprocedural program analysis in software development environments, the problem of call graph construction has received a significant attention [1,4,5,7,11,12,13,14]. A call graph is a static representation of dynamic invocation relationships between procedures (or functions or methods) in a program. A node in this directed graph represents a procedure and an edge $(p \rightarrow q)$ exists if the procedure p can invoke procedure q. In object-oriented languages, a call graph is typically constructed as a side-effect of interprocedural class analysis or type analysis [4]. Interprocedural class analysis computes a set of classes or types for each program variable such that each run-time value bound to a variable is a direct instance of one of the classes bound to the variable. The set of classes associated with the arguments of a dynamically dispatched message send at a callee site determine the set of callee methods or procedures.

A number of techniques have been developed for call graph construction in the last two decades. These techniques not only vary in their cost and precision, but also in the programming languages features they target. Some of the

* This research was supported by NSF CAREER award ACI-9733520

techniques are specifically for object-oriented languages [1,4,5,11,12], some other primarily target procedure-valued pointers in imperative languages [7,13], and others primarily target functional languages [14]. However, a common characteristic of all these techniques is that they perform *exhaustive analysis*, i.e. given a code, they compute the set of procedures that can be called at each of the call-sites in the code.

We believe that with increasing popularity of just-in-time or dynamic compilation and with increasing use of program analysis in software development environments, there is a strong need for *demand-driven call graph analysis* techniques. In a dynamic or just-in-time compilation environments, aggressive compiler analysis and optimizations are applied to selected portions of the code, and not to other less frequently executed or never executed portions of the code. Therefore, the set of procedures called needs to be computed for a small set of call-sites, and not for all the call-sites in the entire program.

Similarly, when program analysis is applied in a software development environment, demand-driven call graph analysis may be preferable to exhaustive analysis. For example, while constructing static program slices [9], the information on the set of procedures called is required only for the call-sites included in the slice and depends upon the slicing criterion used. Similarly, during program analysis for regression testing, only a part of the code needs to be analyzed, and therefore, demand-driven call graph analysis can be significantly quicker than an exhaustive approach.

In this paper, we present a framework for performing demand-driven call graph analysis. We use a variant of the Callahan's Program Summary Graph (PSG) for performing flow-sensitive analysis for reaching definitions. There are three major steps in our techniques. We initially assume a sound call graph using Class Hierarchy Analysis (CHA). Then, we construct a set of *influencing nodes* that influence the set of procedures invoked at the call-sites we are interested in. Finally, we perform propagation analysis on the set of influencing nodes.

The rest of the paper is organized as follows. In Section 2, we formulate the problem. The program representation used for our analysis is presented in Section 3. Our demand-driven analysis technique is presented in Section 4. We conclude in Section 5.

2 Problem Definition

In this section, we revisit the exhaustive call graph analysis problem, and then give our definition of the problem of demand driven call graph construction. We describe the language features our solution targets. We then state the main challenges in performing demand driven analysis for call graph construction and relate it to the existing solutions for exhaustive call graph construction.

2.1 Problem Definition

Consider a program for which the call graph is to be constructed. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of procedures in the given program. Let $\mathcal{P}$ be the power

```
Class A {                      A::P(A *x, A *y) {
    void P(A *x, A *y);            x.Q(y) ;           cs1
    void Q(A* y);                  y.Q(y) ;           cs2
    void R() ;                 }
}                              main() {
Class B: public A {            A* a;
    void P(A *x, A *y) ;       A* b;
    void Q(A *y) ;
    void R() ;                     ...                s3
}                                  a.R() ;            cs3
A::Q(A *y) {                       a = new B ;        s4
    y = new A ;         s1         b = new A ;        s5
}                                  a.P(b,a) ;         cs4
B::Q(A *y) {                       a = new A ;        s6
    y = new B ;         s2         a.P(a,b) ;         cs5
}                              }
```

Fig. 1. Object-oriented program used as a running example in this paper. Definitions of functions A::R, B::R and B::P are not provided here.

set of P. Let $C = \{c_1, c_2, \ldots, c_m\}$ be the set of dynamic call-sites, that is, the call sites for which the set of procedures possibly invoked is not statically known (without non-trivial analysis). The problem of exhaustive call graph analysis is the same as constructing the mapping

$$M \, : \, C \rightarrow \mathcal{P}$$

In other words, we compute, for each call-site in the program , the set of procedures that can be possibly be invoked.

In contrast, the demand-driven call graph analysis involves constructing the set of procedures called at a specific call-site, or a (small) set of call-sites. Given a particular call-site c_i, we compute the mapping from c_i to a subset of P. Formally, this can be expressed as

$$m_i \, : \, c_i \rightarrow \mathcal{P}$$

2.2 Language Model

We now state the language features we focus on. We are interested in treating common object-oriented languages like C++ and Java. At the same time, we are interested in focusing on a set of simple language features, so that we can present the details of our technique with simplicity.

A class has members fields and member methods or procedures. A member procedure `pname` of a class `pclass` is denoted as `pclass::pname`. A base class can be extended by another class. In such a case, the base class is called a *superclass* and the class extending the base class is called a *subclass*. A class can only extend one superclass, treating languages with multiple inheritance is beyond the scope of this paper. The *set of subclasses* of a given class c is constructed transitively by including the class c, and any class that extends c or any class already included in the set. Similarly, the *set of superclasses* of a given class c is constructed transitively by including the class c, the class that the class c extended or any class that is extended by a class already included in the set.

We assume a statically-typed language in which a new object reference is created by a statement of the type `r = new Classname;` . Such a statement gives the type `Classname` to the variable `r`. The type of this instance of the variable `r` can only change as a result of another `new` statement.

The procedure calls are made as *static procedure calls*, in which an explicitly declared procedure is directly called by its name, or as *dynamic procedure calls*. A dynamic procedure call is of the format `r.pname(paramlist)`. We assume that the actual procedure invoked at this call-site depends only upon the type of `r`. If r is of type `Classname`, then the procedure invoked at this call-site is the procedure `pname` defined in the class `Classname` (if it exists) or the procedure with the name `pname` declared in the nearest superclass of `Classname`. However, the type of `r` is usually not known through static declarations. If the static declaration of `r` is of class `Declname`, `r` can be dynamically assigned to an object of any subclass of `Declname`.

An example code that will be used as a running example is shown in Figure 1. We assume that all parameters are passed by reference and all procedures are declared to be virtual (i.e. can be over-written by a derived class). All member methods and fields are assumed to be public for simplicity. The class `A` is the base class, from which the class `B` is derived. Procedures `P`, `Q` and `R` are each defined in both these classes.

2.3 Overview of the Solution

We now give an overview of the problems involved in computing the mapping $m_i : c_i \rightarrow \mathcal{P}$. Let the call-site c_i have the format `r.pname()`.

We first need to compute the types that `r` can possibly have at the program point c_i. Since a new type can only be assigned to `r` by a statement of the type `r = new Classname;`, we need to examine such *instantiations* of `r` that reach c_i. Let p be the statement where c_i is placed. There are three possibilities for each reaching instantiation of `r`:

1. It is defined in the procedure p.
2. The procedure p has another call-site c_j, such that the instantiation of `r` reaching c_i is passed as a reference parameter at the call-site c_j.
3. The instantiation of `r` reaching c_i is a formal parameter for the procedure p.

Case 1: This case is trivial because the reaching types of `r` can be determined by intraprocedural analysis within the procedure p. For example, in Figure˜refex:code, the dynamic calls at call-sites `cs4` and `cs5` can be resolved by intraprocedural propagation. The following non-trivial analysis is required for cases 2 and 3.

Case 2: We need to analyze the procedures that can possibly be invoked at the call-site c_j to be able to determine the type of `r`. This is relatively easy if the call-site c_j is a static call-site. However, if the call-site c_j is a dynamic call-site, we need to perform additional analysis to be able to determine the set of procedures invoked at the call-site c_j, and then need to perform reaching definitions analysis in the procedures invoked at c_j. For example, in Figure 1, for resolving the call-site `cs2`, we need to look at the possible values of the reference parameter `y` at the end of the procedure calls possible at the call-site `cs1`.

Case 3: To determine the types associated with `r` when entering the procedure p, we need to analyze all the call-sites at the which the procedure p is invoked. In the absence of the call graph, however, we do not know the set of call-sites that can invoke the procedure p. Therefore, we need to start with a conservative estimate of the set of call-sites at which the procedure p may be invoked. Analyzing these call-sites, we need to determine:

– Can the procedure p be invoked at this call-site ?
– If so, what are the types associated with the actual parameter at this call-site corresponding to the formal parameter `r` ?

For example, in Figure 1, for resolving the call-site `cs1`, we need to look the possible values of the reference parameter `x` at the entry of the proceedure `A::P`.

The previous work in demand-driven flow-sensitive data-flow analysis [6,8] (which does not apply to call graph analysis) formulates the problem as that of *query propagation*. The initial data-flow problem to be solved on a demand-basis is itself a query, whose solution is described in terms of queries at successor or predecessor nodes.

We can also describe the analysis for cases 2 and 3 as query propagation problems. Consider initially the case 2. If the call-site c_j is a dynamic call site, we need to answer the query

$$m_j \ : \ c_j \rightarrow \mathcal{P}$$

to answer our initial query. Irrespective of whether the call-site c_i is a dynamic call-site or a static call-site, we will need to know the types associated with `r` at the end of the call to each of the procedures that can possibly be invoked at this call-site.

The query propagation for case 3 can be stated as follows. We need to determine the set of call-sites C', such that for each

$$\forall\, c_k \in C' \quad (m_k \ : \ c_k \rightarrow \mathcal{P}_k) \ \rightarrow \ p \in \mathcal{P}_k$$

2.4 Overview of Call Graph Analysis Techniques

For ease in explaining our demand-driven call graph construction technique, we describe an existing classification of call graph analysis techniques [4]. All possible call graphs constructued by various algorithms, including the initial call graphs assumed and the results after each stage of the algorithm, can be described by a lattice. The *top-most* element of the lattice (denoted by $G_\top$) has no edges between any of the procedures. The *bottom-most* element of the lattice is denoted by $G_\perp$ and includes edges to all procedures in the entire program at each call-site. An *ideal graph* is the one in which each edge corresponds to an actual procedure invocation during one of the possible executions of the program. Obviously, the ideal graph is uncomputable.

Actual call graphs at each stage in various algorithms can be classified as *optimistic* or *sound*. An optimistic graph has fewer edges than the ideal graph, i.e., additional edges need to be inserted to model invocations during actual executions. A sound graph has more edges than ideal graph, so some edges can be deleted to improve the accuracy of the analysis. The process of adding more edges to an optimistic graph (to move it down the lattice) is referred to as the *monotonic refinement*. The process of deleting some edges from a sound graph (to move it up the lattice) is referred to as *non-monotonic improvement*.

3 Program Summary Graph

We use the interprocedural representation Program Summary Graph (PSG), initially proposed by Callahan [2], for presenting our demand-driven call graph analysis technique. We initially give the original definition of PSG and then describe several extensions of PSG that we use for our purpose.

A program summary graph is a representation of the complete program that is much concise as compared to the Myers' Supergraph or Interprocedural Control Flow Graph (ICFG) [10], but is more detailed than a call graph, and allows *flow-sensitive* interprocedural analysis.

Data-flow within procedures and between procedures is represented through edges between nodes of the following four types:

- *Entry nodes*; there is one entry node for each formal parameter of each procedure.
- *Exit nodes*; there is one exit node for each formal parameter of each procedure.
- *Call nodes*; there is one call node for each actual parameter at each call-site.
- *Return nodes*; there is one return node for each actual parameter at each call-site.

Edges between these nodes can be classified as *intraprocedural* and *interprocedural*. Intraprocedural edges summarize data-flow within the procedures. These edges are inserted after solving the standard data-flow problem of reaching definitions within each procedure. Specifically, the intraprocedural edges are inserted:

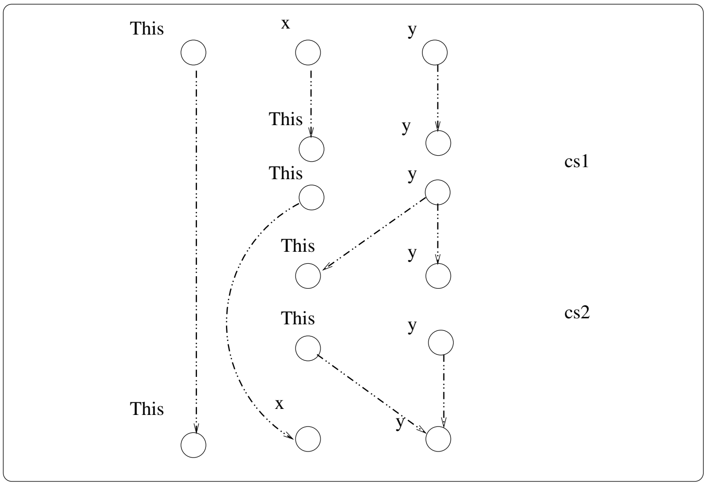

Fig. 2. Procedure `A::P`'s portion of PSG

- From an entry node to a call node if the value of the corresponding formal parameter at the procedure entry reaches the corresponding actual parameter at the call-site.
- From an entry node to an exit node if the value of the corresponding formal parameter at the procedure entry reaches a procedure return statement.
- From a return node to a call node if the value of the corresponding actual parameter at the call return reaches the corresponding actual parameter at the call node.
- From a return node to an exit node if the value of the corresponding actual parameter at call return reaches a procedure return statement.

Interprocedural edges in the graph represent bindings of actual parameters to the formal parameters and vice-versa. Specifically, interprocedural edges are inserted:

- From a call node to an entry node to represent the binding of an actual parameter at the call-site to the formal parameter at procedure entry.
- From an exit node to a return node to represent the binding of a formal parameter at the procedure exit to the actual parameter at the call return.

To perform demand-driven call graph analysis, we make the following three extensions to the original definition construction method of PSG.

This pointer as a parameter: Consider a call site of the form `r->pname(paramlist)`. For our analysis, besides having call and return nodes corresponding to each actual parameter in the list `paramlist`, we also need to have a call and return node for the object reference `r`. We refer to such nodes as nodes for THIS pointer, consistent with the C++ terminology. We also insert one THIS pointer node each at procedure entry and return for each procedure.

Internal Intraprocedural Nodes: We insert new *internal* intraprocedural nodes in the graph besides entry, exit, call and return nodes. These intraprocedural nodes represent statement that provide a new type to a variable. In the language model we assumed, these are statements of the type `r = new Classname`. New intraprocedural edges are inserted from internal nodes as follows. If a definition given in an internal nodes reaches an actual parameter at a call-site, a new edges is inserted from the internal node to the corresponding call node. If a definition given in an internal node reaches a procedure return statement, a new edge is inserted from the internal node to the corresponding exit node.

Demand Driven Construction: The Program Summary Graph is not fully constructed at the beginning of the analysis, since it will require knowing the call graph. Instead, the program summary graph is constructed on a demand basis. If v is the call node for which we are determining the types, we perform a reachability analysis on the portion of the graph constructed and check if v is reachable from one of the nodes of a procedure p. If so, the CFG of the procedure p is constructed, and intraprocedural edges of the procedure p's portion of PSG are inserted. The portion of PSG constructed in performing the propagation analysis is referred to as the Partial Program Summary Graph (PPSG) and set of nodes in the PPSG is referred to as the *set of influencing nodes*.

Procedure `A::P`'s portion of PSG is shown in Figure 2.

4 Demand-Driven Call Graph Construction

In this section, we describe the various phases of our algorithm for constructing call graph on a demand-driven basis and analyze the complexity of our technique.

4.1 Overview of the Algorithm

We are considering the problem of computing the set of procedures that can be invoked at a call-site c_i, of the form `r->pname()`. The most important goal for our algorithm is to analyze as few procedures as possible while determining the set of procedures that can be invoked at the call-site c_i. Our algorithm initially assumes a sound or conservative call graph, i.e., one with much larger number of edges than what can be taken during actual executions. This initial sound call graph is constructed using the results of Class Hierarchy Analysis (CHA) [3]. Such an initial sound call graph is also not constructed explicitly for the entire program, but is constructed on a demand basis. Each procedure's components are added only after it is known that this procedure may influence the types associated with the object reference `r` at the call-site c_i.

There are two main phases in our algorithm. Initially, we perform reachability analysis using the sound call graph to determine the set of influencing nodes and to construct the Partial Program Summary Graph (PPSG). The second phase involves performing data-flow propagation on the PPSG to improve the precision of the call graph using reaching types information.

4.2 Initial Sound Call Graph

As we described earlier in Section 2, call graph analysis techniques either start from an optimistic graph and add more edges, or can start from a sound or conservative graph and delete edges. We prefer to start from a sound graph and delete edges since it allows us to compute the set of influencing nodes. One possible initial sound graph is $G_{\perp}$, in which each call-site can invoke all procedures. Obviously, starting from such an initial call graph will defeat the purpose of demand-driven analysis as all the nodes in the entire program will get added to the set of influencing nodes. We can construct a relatively accurate initial call graph by performing relatively inexpensive Class Hierarchy Analysis (CHA) [3].

CHA involves having knowledge of all the classes declared in the entire program, including which class extends another class, and the set of procedures declared in each of the classes. Consider a call-site of the form `r->rname()`, such that the declared type of `r` is `rclass`. Let $\mathcal{R}$ be the set of subclasses of `rclass`. For each class in the set $\mathcal{R}$, we determine the nearest superclass (including itself) in which a procedure with the name `rname` is declared. Let us denote such a set of classes by $\mathcal{R}'$. Then, as a result of class hierarchy analysis, we know that the possible procedures that can be called at this call-site are of the form `pclass::rname`, where `pclass` belongs to the set $\mathcal{R}'$.

Alternatively, consider a procedure p of the form `pclass::pname`. By knowing all class declarations in the program, we can determine the set of subclasses of `pclass`. By further examining the procedures declared in each of these classes, we can narrow this set down to classes for which `pclass` is the earliest superclass for which the procedure `pname` is defined. Let us denote such a set by $\mathcal{S}$. The procedure p can be called at any dynamic call-site of the form `r->pname` where the declared type of the reference `r` belongs to the set $\mathcal{S}$.

4.3 Preliminary Definitions

In presenting our technique, we use the following definitions.

pred(v) : The set of predecessors of the node v in the PPSG. This set is initially defined during the construction of PPSG and is not modified as the type information becomes more precise.

proc(v) : This relation is only defined if the node v is an entry node or an exit node. It denotes the name of the procedure to which this node belongs.

TYPES(v): The set of types associated with a node v in the PSG during any stage in the analysis. This set is initially constructed using Class Hierarchy Analysis, and is later refined through data-flow propagation.

THIS_NODE(v): This is the node corresponding to the THIS pointer at the procedure entry (if v is an entry node), procedure exit (if v is an exit node), procedure call (if v is a call node) or call return (if v is a return node).

THIS_TYPE(v): If the vertex v is a call node or a return node, THIS_TYPE(v) returns the types currently associated with the call node for the THIS pointer at this call-site. This relation is not defined if v is an entry or exit node.

PROCS(S): Let S be the set of types associated with a call node for a THIS pointer. Then, PROCS(S) is the set of procedures that can actually be invoked at this call-site. This function is computed using Class Hierarchy Analysis (CHA).

```
Let $v$ be the this pointer node at the call site
Let $p$ be the procedure to which $v$ belongs
Initialize $Workset$ to $\{v\}$
Initialize $Procset$ to $\{p\}$
Initialize all nodes to be not *marked*
$Construct\_PSG\_Portion(p)$

*While* $Workset$ is not empty
    Select and remove vertex $u$ from $Workset$
    **case (type of u):**
        **call or exit:**
            *foreach* predecessor $w$ of $u$
                *If* $w$ is not *marked*
                    $Workset = Workset \cup \{w\}$
        **return:**
            *If* THIS_NODE($u$) is not *marked*
                $Workset = Workset \cup$ THIS_NODE($u$)
            *foreach* possibly called function $q$
                *If* $q \notin Procset$
                    $Procset = Procset \cup \{q\}$
                    $Construct\_PSG\_Portion(q)$
            *foreach* predecessor $w$ of $u$
                *If* $w$ is not *marked*
                    $Workset = Workset \cup \{w\}$
        **entry:**
            $Workset = Workset \cup$ THIS_NODE($u$)
            *foreach* possible callee function $q$
                *If* $q \notin Procset$
                    $Procset = Procset \cup \{q\}$
                    $Construct\_PSG\_Portion(q)$
            *foreach* predecessor $w$ of $u$
                *If* $w$ is not *marked*
                    $Workset = Workset \cup \{w\}$
```

Fig. 3. Constructing the Partial Program Summary Graph (PPSG)

4.4 Constructing the Set of Influencing Nodes

We now describe how we compute the set of nodes in the PSG for the entire program that influence the set of procedures invoked at the given call-site c_i.

The PSG for the entire program is never constructed. However, for ease in presenting the definition of the set of influencing nodes, we assume that the PSG components of all procedures in the entire program are connected based upon the initial sound call graph.

Let v be the call node for the THIS pointer at the call-site c_i. Given the hypothetical complete PSG, the set of influencing nodes (which we denote by S) is the minimal set of nodes such that:

- $v \in S$
- $(x \in S) \wedge (y \in pred(x)) \rightarrow y \in S$
- $x \in S \rightarrow \text{THIS\_NODE}(x) \in S$

Starting from the node v, we include the predecessors of any node already in the set, till we reach internal nodes that do not have any predecessors. For any node included in the set, we also include the corresponding node for the THIS pointer (denoted by THIS_NODE) in the set.

Such a set of influencing node and the partial PSG can be constructed by an iterative algorithm, which is shown in Figure 3. Two main data-structures maintained in the algorithm are $Workset$ and *Procset*. $Workset$ is the set of nodes whose predecessors have not been analyzed yet. *Procset* is the set of procedures that have been analyzed and whose portions of the PSG has been constructed.

The algorithm progresses by removing a node from the $Workset$. If this node is a call or exit node, all the predecessors of this node are within the same procedure. These predecessors are added to the $Workset$. If the node u (removed from the $Workset$) is a return node, the predecessors of this node are the exit nodes of the procedures that can be invoked at this call-site. Such a set of procedures is known (not necessarily accurately) from our construction of the initial sound call graph. Let q be any such procedure. If q is not in the set *Procset* (i.e. it has not been analyzed yet), then the function *Construct_PSG_Portion*(q) is invoked. This function analyzes the CFG of the procedure q to construct its portion of the PSG. For each callee of q that has been analyzed, edges from its call nodes to entry nodes of q and edges from exit nodes of q to its return nodes are inserted. Similarly, for each function called by q that has been analyzed, we insert edges from call nodes at q to its entry nodes from its exit nodes to the return nodes at q. After all such procedures called at this call-site have been analyzed and edges have been inserted, we add the predecessors of the node u to the $Workset$.

The edges inserted at these call-sites are obviously based upon an initial sound call graph, that needs to be refined by our analysis. For this purpose, we need to know the types associated with the THIS_NODE at the call-site corresponding to the return node. For this reason, we also add THIS_NODE(u) to the $Workset$.

The actions taken for an entry node are very similar to the actions taken for a return node. The only difference is that instead of analyzing the procedures that can be called at that call-site, we analyze the procedures that have a call-site that can invoke this procedure.

4.5 Call Graph Refinement

The next step in the algorithm is to perform iterative analysis over the set of nodes in the Partial Program Summary Graph (PPSG) to compute the set of types associated with a given initial node. This problem can be modeled as computing the data-flow set TYPES with each node in the PPSG and refining it iteratively. The initial values of TYPES(v) are computed through class hierarchy analysis that we described earlier in this section. If a formal or actual parameter is declared to be a reference to class `cname`, then the actual runtime type of that parameter can be any of the subclasses (including itself) of `cname`.

The refinement stage can be described by a single equation, which is shown in Figure 4. Consider a node v in PPSG. Depending upon the type of v, three cases are possible in performing the update:

1. v is a call or exit node,
2. v is an entry node, and
3. v is a return node.

In Case 1., the predecessors of the node v are the internal nodes, the entry nodes for the same procedure, or the return nodes at one of the call-sites within this procedure. The important observation is that such a set of predecessors does not change as the type information is made more precise. So, the set TYPES(v) is updated by taking union over the sets of TYPES(v) over the predecessors of the node v.

We next consider case 2, i.e., when the node v is an entry node. $proc(v)$ is the procedure to which the node v belongs. The predecessors of such a node are call nodes at all call-sites at which the function $proc(v)$ can possibly be called, as per the initial call graph assumed by performing class hierarchy analysis. Such a set of possible call-sites for $proc(v)$ gets restricted as interprocedural type propagation is performed. Let p be a call node that is a predecessor of v. We want to use the set TYPES(p) in updating TYPES(v) only if the call-site corresponding to p invokes $proc(v)$. We determine this by checking the condition $proc(v) \in$ PROCS(THIS_TYPE(p)). The function THIS_TYPE(p) determines the types currently associated with the THIS pointer at the call-site corresponding to p and the function PROCS determines the set of procedures that can be called at this call-site based upon this type information.

Case 3 is very similar to the case 2. If the node v is a return node, the predecessor node p to v is an exit node. We want to use the set TYPES(p) in updating TYPES(v) only if the call-site corresponding to v can invoke the function $proc(p)$. We determine this by checking the condition $proc(p) \in$ PROCS(THIS_TYPE(v)). The function THIS_TYPE(v) determines the types currently associated with the THIS pointer at the call-site corresponding to v and the function PROCS determines the set of procedures that can be called at this call-site based upon this type information.

$$\text{TYPES}(v) = \begin{cases} \text{TYPES}(v) \bigcap (\bigcup_{p \,\in\, pred(v)} \ \text{TYPES}(p)\,) \\ \qquad\qquad\qquad \text{if v is call or exit node} \\ \text{TYPES}(v) \bigcap (\bigcup_{(p \in pred(v)) \wedge (proc(v) \in \text{PROCS}(\text{THIS\_TYPE}_{(p)}))} \text{TYPES}(p)) \\ \qquad\qquad\qquad \text{if v is an entry node} \\ \text{TYPES}(v) \bigcap (\bigcup_{(p \in pred(v)) \wedge (proc(p) \in \text{PROCS}(\text{THIS\_TYPE}_{(v)}))} \text{TYPES}(p)) \\ \qquad\qquad\qquad \text{if v is a return node} \end{cases}$$

Fig. 4. Data-Flow Equation for Propagating Type Information

4.6 Complexity Analysis

We now calculate the worst-case complexity of the technique proposed earlier. We initially analyze the complexity of the phase for creating the PPSG and then analyze the complexity of the call graph refinement phase. We define the following terms for our presentation:

N	:	The number of nodes in PPSG
E	:	The number of edges in the PPSG
D	:	The maximum depth of a class hieararchy

Creating PPSG: The algorithm for constructing the set of influencing nodes was presented in Figure 3. We are focussing on analysis of large codes comprising a very large number of procedures. Therefore, we assume that the cost of constructing and analyzing a single procedure is small and the function $Construct\_PSG\_Portion$ can be executed in a constant time. Then the overall cost of the execution of the algorithm presented in Figure 3 is $O(N + E)$. This is because each node is removed once from the $Workset$ (and the total number of elements ever added in the $Workset$ is N) and each edge is followed once to consider the source of the edge.

Iterative Analysis: The iterative analysis is performed over a graph with N nodes and E edges. The maximum number of times that the types associated with a node can change is bounded by D, the maximum depth of the class hierarchy. Since the total number of nodes in the graph is N, the maximum possible length of any acyclic path in the graph is also N. Therefore, the iterative analysis needs to be repeated over a maximum of a $N \times D$ steps. Each step in the iterative analysis requires updating the types of N nodes, which will take $O(N)$ time, assuming that the number of predecessors of a node is bounded by a small constant. Therefore, the complexity of iterative analysis is $O(N^2D)$.

The efficiency of our technique depends upon the size of the PPSG constructed. If the PPSG is almost as large as the complete PSG for the entire program, our algorithm basically reduces to the classical 0-CFA construction technique proposed by Shivers [14]. If PPSG is significantly smaller than the

PSG for the entire program, our technique is very efficient as compared to the exhaustive techniques.

4.7 Theoretical Results

As we mentioned in the previous subsection, our analysis is similar to Shiver's 0-CFA, except that we perform demand-driven analysis using propagation only on PPSG, whereas Shiver's 0-CFA is an exhaustive technique requiring propagation over the PSG for the entire program. One important issue is the accuracy of our demand driven technique as compared to the similar exhaustive technique. We show below that the accuracy of our demand-driven technique is the same as the accuracy of Shiver's 0-CFA technique.

Theorem 1 *The sets* TYPES(v) *computed by our technique for any node v in the set S is identical to the set constructed by 0-CFA exhaustive technique.*

Proof: The basic idea behind the proof is as follows. Any node that can influence the value of TYPES(v) is included in the set S. Therefore, a propagation just over the set S results in as accurate results as propagating over the entire PSG.

There are two interesting consequences of this theorem. Suppose, we initially computed the types for a node v by performing demand-driven analysis and the set of influencing nodes S that is computed includes a node w. If we later require the type information for the node w, we can just use the information computed while performing demand-driven analysis for node v. Further, suppose we need to compute types information for a node z. While computing the set of influencing nodes, we include a node y, which was previously included in a set of influencing nodes. We can just use the previously computed types information for y, and do not need to continue to include predecessors of y in the new set of influencing nodes.

5 Conclusions

For performing interprocedural program analysis and transformations on programs with polymorphism, we need to know which methods can be called at each of the call-sites. In a number of scenarios, it can be important to compute this information on a demand-basis for certain call-sites only, rather than the entire program. Examples of such scenarios are 1) performing optimizations during just-in-time or dynamic compilation, 2) performing program analysis (like slicing) for software development, and 3) performing scalable analysis for very large programs.

In this paper we have presented, to the best of our knowledge, first solution to the problem of demand-driven resolution of call-sites in object-oriented programs. Our technique is based upon computing a set of influencing nodes, and then applying data-flow propagation over such a set.

We have two major theoretical results. We have shown that the worst-case complexity of our analysis is the same as the well known 0-CFA exhaustive analysis technique, expect that our input is cardinality of the set of influencing nodes, rather than the total number of nodes in the graph. Thus, the advantage of our demand-driven technique depends upon the relative size of set of influencing nodes and the total number of nodes. Second, we have shown that they type information computed by our technique for all the nodes in the set of influencing nodes is as accurate as the 0-CFA exhaustive analysis technique.

References

1. David Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 324–341, October 1996. 125, 126
2. D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988. 130
3. Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 93–102, La Jolla, California, 18–21 June 1995. *SIGPLAN Notices* 30(6), June 1995. 132, 133
4. Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings of the POPL'98 Conference*, 1998. 125, 126, 130
5. Amer Diwan, J. Elliot Moss, and K. Mckinley. Simple and effective analysis of statically typed object-oriented programs. In *Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 292–305, October 1996. 125, 126
6. Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedual data flow. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, California, January 1995. 129
7. Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992. 125, 126
8. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *In SIGSOFT '95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, 1995. 129
9. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990. 126
10. E. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981. 130
11. Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 367–378, San Francisco, California, January 1995. 125, 126

12. Hemant Pande and Barbara Ryder. Data-flow-based virtual function resolution. In *Proceedings of the Third International Static Analysis Symposium*, 1996. 125, 126
13. B. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979. 125, 126
14. O. Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 190–198, New Haven, CN, June 1991. 125, 126, 137