Efficient Computation of Interprocedural Control Dependence

James Ezick
Cornell University
4139 Upson Hall
Ithaca, NY 14853 USA
(607) 255-4934
ezick@cs.cornell.edu

Gianfranco Bilardi
Università di Padova
Dip. di Elettronica ed Informatica
Via Gradenigo, 6A
35131 Padova, Italy
+390498277819
bilardi@dei.unipd.it

Keshav Pingali
Cornell University
457A Rhodes Hall
Ithaca, NY 14853 USA
(607) 255-7203
pingali@cs.cornell.edu

Abstract

Control dependence information is useful for a wide range of software maintenance and testing tasks. For example, program slicers use it to determine statements and predicates that might affect the value of a particular variable at a particular program location. In the intraprocedural context an optimal algorithm is known for computing control dependence which unfortunately relies critically on the underlying intraprocedural postdominance relation being tree-structured. Hence, this algorithm is not directly applicable to the interprocedural case, where the transitive reduction of the postdominance relation can be a directed acyclic graph (DAG), with nodes having multiple immediate dominators.

In this paper, we present an efficient, conceptually simple algorithm for computing the interprocedural postdominance relation, and use it to compute interprocedural control dependence. For an interprocedural control flow graph $G = (V, E)$, our algorithm takes time and space $O(|V|^2 + |V||E|)$. Unlike other algorithms, ours does not perform confluence operations on whole bit-vectors, hence allowing it to scale to larger programs.

Keywords

Interprocedural Analysis, Postdominance, Control Dependence, Reachability

1 Introduction

Many problems in software engineering such as testing and maintenance of programs require the computation of the *control dependence relation* of the program. Intuitively, a statement w is control dependent on a statement u if there are multiple exits out of u and the choice of exit determines whether w is executed. For example in an *if-then-else* construct, statements on the two sides of the conditional statement are control dependent on the predicate.

The most common use of control dependence in software engineering is in determining whether a change to the semantics of a program statement affects the execution of another program statement. Like most program analysis problems, this one is undecidable, but computing program dependences provides a reasonable approximate approach to answering such a question. The research community has focused much attention on such *slicing tools* [15, 20, 31]. For example, the *system dependence graph* (SDG) [15], an interprocedural extension of the program dependence graph [8, 23], incorporates edges for both control and data dependence to allow programs to be sliced at a point p with respect to a variable x defined or used at p.

In restructuring and optimizing compilers, control dependence is used in scheduling instructions across basic-block boundaries for speculative or predicated execution [2, 9, 22], in merging program versions [14], and in automatic parallelization [1, 8, 30]. In some applications such as code scheduling, it is necessary to know which nodes have the same control dependences as a given node. This information is useful in code scheduling because basic blocks with the same control dependences can be treated as one large basic block, as is done in region scheduling [11]. This information can also be used to decompose the control flow graph of a program into single-entry single-exit (SESE) regions, and this decomposition can be exploited to speed up dataflow analysis by combining structural and fixpoint induction [17, 18] and to perform dataflow analysis in parallel [10, 18].

Formulating a precise definition of control dependence in programs with nested control structures, multiway branches, unstructured flow of control, and procedure calls can be quite subtle. The most commonly used definition, due to Ferrante *et al*, is based on the graph-theoretic concept of *postdominance* [8]. This work was later extended by Podgurski and Clarke who distinguished between several notions of control dependence [25]. Bilardi and Pingali [3], proposed a generalized framework to unify many different such notions. Most of the research in this area has focused on computing *intraprocedural* control dependence [8, 24, 6, 26, 27]. In this approach, each procedure is treated in isolation, by ignoring transfer of control due to procedure calls and returns. While adequate for some applications such as instruction scheduling, ignoring calls and returns is not an option for other applications such as interprocedural dataflow analysis. A popular method to perform intraprocedural dataflow analysis consists in building *sparse dataflow representations* (such as the Static Single Assignment (SSA) form [6], sparse dataflow evaluator graphs [5], and Quick Propagation Graphs [18]), solving dataflow problems in these representations, and then projecting the solution onto the original program. The construction of these sparse representations is based on computing control dependence in the reverse control graph of the program. Extending this method to the interprocedural context obviously requires the computation of interprocedural control dependence.

In the intraprocedural case, a key step toward control dependence computation is the construction of the transitive reduction of the postdominance relation. This reduced relation is tree-structured, a fact that is crucially exploited both in computing it [19, 4] and in using it for control dependence computations [24]. In the interprocedural case, the transitive reduction of postdominance is not necessarily a tree, so that these intraprocedural algorithms can no longer be used. The key point, to be discussed more closely in Section 2, is that while all graph-theoretic paths in the intraprocedural control flow graphs correspond to legal program executions, this is no longer true in the interprocedural case. Indeed, a procedure can be called from any number of different program points, but at the end of the call control can return only to the program point just after the point of invocation.

A search of the literature revealed only a handful of algorithms for computing interprocedural control dependence [13]. Loyall and Mathisen [21] give an algorithm which has been shown to be incorrect by [13]. Harrold, Rothermel and Sinha present an interprocedural control dependence algorithm with $O(N^5)$ complexity [12]. In later work [31], they present a method for augmenting control flow graphs to handle programs with arbitrary control flow.

In this paper, we present a new and practical algorithm for computing the interprocedural postdominance relation of a program, and use it to compute the interprocedural control dependence relation. Our postdominance algorithm, introduced in Section 3, is based on determining reachability of nodes in the interprocedural control flow graph along so-called *valid path suffixes*, defined in Section 2, which reflect the constraints of legal program executions. The running time of our algorithm is $O(|V|(|E| + |V|))$ where $|V|$ and $|E|$ are, respectively, the number of nodes and of edges in the interprocedural control flow graph. Precomputing and caching certain sets of reachable nodes improves the efficiency of our algorithm in practice, as we show in Section 4. Our algorithm produces the full postdominance relation, which is transitive. In Section 5, we show how, by means of a single boolean matrix multiplication, one can obtain the transitive reduction of postdominance, which turns out to be preferable to work with in some applications. In Section 6, we outline how to compute the interprocedural control dependence relation of the program. We give experimental results in Section 7, comparing the performance of our algorithm with that of an iterative algorithm based on dataflow analysis. Finally, we discuss ongoing work in Section 8.

2 Concepts and Definitions

A program is formed by a family $\mathcal{P}$ of procedures which can call each other, including a distinguished procedure `MAIN` which no other procedure may call. The *interprocedural* control-flow graph (ICFG) of a program models the possible transfer of control both between statements of the same procedure and between different procedures, by means of the call-return mechanism [29]. As a preliminary step, it is convenient to introduce the *intraprocedural* control-flow graph (iCFG) of an individual procedure.

Definition 1 *A* **intraprocedural control-flow graph (iCGF)** *of a procedure P is a directed graph $G_P = (V_P, E_P)$ in which nodes represent statements and an edge $u \rightarrow v$ represents possible flow of control from statement u to statement v. The node set V_P can be partitioned as*

$$V_P = \{\mathtt{START}_P\} + \{\mathtt{END}_P\} + V_P^c + V_P^r + V_P^l$$

where: $\mathtt{START}_P$ is a node with no predecessors from which every node is reachable; $\mathtt{END}_P$ is a node with no successors and reachable from every node; V_P^c is the set of call *nodes, which have exactly one outgoing edge; V_P^r is the set of* return *nodes, which have exactly one incoming edge; $|V_P^c| = |V_P^r|$.*

The edge set can be partitioned as $E_P = E_P^i \cup E_P^s$. An internal *edge $(u, v) \in E_P^i$ corresponds to direct transfer of control from u to v, internally to procedure P.*

The set E_P^s of short-cut *edges is a subset of $V_P^c \times V_P^r$ and induces a one-to-one correspondence between call nodes and return nodes. For each $(u, v) \in E_P^s$, a $label(u, v) \in \mathcal{P}$ identifies the procedure being called at u and returning control at v. We shall use the convenient notations $v = r(u)$ and $u = c(v)$. If $label(u, v) = F$, we shall say that u is a* call node *for F and v is a* return node *from F.*

The ICFG is obtained by assembling the iCFGs of all procedures and replacing each short-cut edge (u, v) having $label(u, v) = F$ with two interprocedural edges $(u, \mathtt{START}_F)$ and $(\mathtt{END}_F, v)$, which are said *to correspond* to each other.

Definition 2 *The* **interprocedural CFG (ICFG)** *of a program is a graph $G = (V, E)$ with $V = \cup_{P\in\mathcal{P}} V_P$ and $E = E^i + E^c + E^r$, where $E^i = \cup_{P\in\mathcal{P}} E^i_P$ is called the set of* internal *edges, and E^c and E^r, defined as*

$$E^c = \cup_{P\in\mathcal{P}}\{(u, \mathtt{START}_F) : (u, v) \in E^s_P,\ label(u, v) = F\},$$

$$E^r = \cup_{P\in\mathcal{P}}\{(\mathtt{END}_F, v) : (u, v) \in E^s_P,\ label(u, v) = F\}$$

and called the sets of the call *edges and of the* return *edges, respectively.*

Figure 1(a) shows an interprocedural CFG for a simple program with two procedures `MAIN` and `F` in which `F` is called from two places in `MAIN`.

Not every ICFG path corresponds to a possible path of execution since a procedure can be invoked from many call sites but each invocation can only return to its corresponding call site. For example, in Figure 1(a), the path `(a,b,c,j,k,l,f,h,i)` does not correspond to a valid path of execution since the call to `F` at node `c` does not return to node `e`. This intuition is captured by defining a *valid path* (this is called a *complete* valid path by Sharir and Pnueli [29]) as a path where the subsequence of call and return edges is *proper*, in the following sense.

Definition 3 *The set of* **proper sequences** *of call and return edges is defined as the context-free language on the alphabet $(E^c + E^r)$ which (i) contains the empty sequence, (ii) is closed under concatenation, and (iii) if it contains sequence σ then it also contains those sequences of the form $(u, \mathtt{START}_F)\sigma(\mathtt{END}_F, v)$, for each short-cut edge (u, v) with $label(u, v) = F$.*

Definition 4 *A path from $\mathtt{START}_{\mathtt{MAIN}}$ to $\mathtt{END}_{\mathtt{MAIN}}$ in the ICFG G of a program, viewed as a sequence of edges, is a* **valid path** *if its subsequence of call and return edges is proper.*

In the remainder of this paper, we shall restrict our attention to ICFGs where each node v occurs on some valid path. We will have occasion to deal with the following kind of valid path segment, called *same-level* valid path segments in the literature [28].

Definition 5 *A* **same-level** *valid path segment is a sequence of edges which appear consecutively in some valid path, whose first and last nodes belong to the same procedure, and whose subsequence of call and return edges is proper.*

In our running example, path `(b,c,j,k,l,e)` is a same-level valid path segment.

The following lemma is needed in the proof of correctness of the algorithm for computing interprocedural postdominance.

Lemma 1 *Let σ_1 and σ_2 be two same-level valid path segments from node u to node v. If $\pi = \pi_1\sigma_1\pi_2$ is a valid path, then so is $\tau = \pi_1\sigma_2\pi_2$.*

Proof: It is easy to see that τ is a path from the start of the program to its end. By Definition 5, the subsequences of call and return edges in both σ_1 and σ_2 are proper sequences. Furthermore, given a proper sequence of call/return edges, we can always replace a proper subsequence with another proper subsequence to obtain a proper sequence. Therefore, τ is a valid path. □

We can now define interprocedural dominance and postdominance between nodes of the ICFG.

Definition 6 *A node v is said to* **interprocedurally dominate** *a node w if v occurs before w in every valid path that contains w.*

A node v is said to **interprocedurally postdominate** *a node w if v occurs after w in every valid path that contains w.*

It is easy to show that dominance and postdominance are transitive relations. Figure 1(b) shows the transitive reduction of the postdominance relation of the program of Figure 1(a).

Control dependence can be introduced formally as follows [8]:

Definition 7 *A node $w \in V$ is said to be* **control dependent** *on edge $(u \to v) \in E$ if*

1. *w postdominates v, and*
2. *if $w \neq u$, then w does not postdominate u.*

By convention, nothing is control dependent on a return edge since the `END` *of a procedure is not a decision point although there may be multiple edges emanating from it.*

Intuitively, if control flows from node u to node v along edge $u \to v$, it will eventually reach node w; however, control may reach `END` from u without passing through w. Thus, u is a 'decision-point' that influences the execution of w.

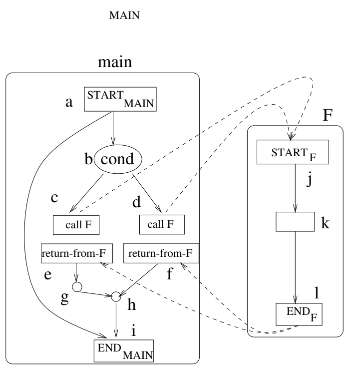
Definition 8 *Given an ICFG $G = (V, E)$, its* **control dependence relation** *is the set $C \subseteq E \times V$ of all pairs (e, w) such that node w is control dependent on edge e.*

The control dependence relation for the program of Figure 1(a) is shown in Figure 1(c).

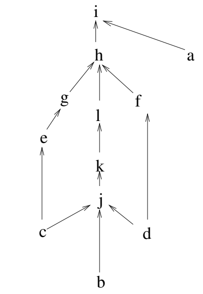
Typically, both software engineering and compiler applications of control dependence require the computation of the following sets derived from C [7]:

Definition 9 *Given a node w and an edge e in a control program graph with control dependence relation C, we define the following* **control dependence sets***:*

- $\mathtt{cd}(e) = \{w \in V : (e, w) \in C\}$,
- $\mathtt{conds}(w) = \{e \in E : (e, w) \in C\}$, *and*
- $\mathtt{cdequiv}(w) = \{v \in V : \mathtt{conds}(v) = \mathtt{conds}(w)\}$.



(a) Interprocedural Control Flow Graph



(b) Interprocedural Postdominator DAG

Edge	Control Dependent Nodes
a → b	{b,j,k,l,h}
b → c	{c,e,g}
b → d	{d,f}

(c) Control Dependence Relation

Figure 1: A Small Program, and its Postdominator DAG and Control Dependence Relation

3 Two Algorithms for Computing Interprocedural Postdominance

In this section, we present two algorithms for computing the interprocedural postdominance relation. The first algorithm is a relatively straight-forward dataflow algorithm that solves a set of monotone dataflow equations iteratively. The second algorithm is based on computing graph-reachability in subgraphs of the interprocedural CFG.

3.1 Iterative Algorithm

It is well-known that the intraprocedural postdominance relation can be computed by solving a set of monotone dataflow equations [19]. This is true for the interprocedural postdominance relation as well. The lattice underlying the equations is the powerset of nodes in the program, ordered by containment, in which the least element is the empty set. The postdominance relation can be described implicitly by writing down a set of equations, one per node, in which the postdominators of a node are expressed as a function of the postdominators of its successors in the interprocedural CFG and some other nodes. Figure 2 shows the rules for generating these equations. In general, the postdominators of a node n are n itself, and any other node that postdominates all the control flow successors of n. A `Call` node is also postdominated by any node that postdominates its corresponding `Return` node.

This is obviously a monotone dataflow problem, and its solution can be found iteratively by starting with the initial approximation in which the postdominator set of each node is assumed to be the set of all nodes in the program, and repeatedly refining this approximation by substituting into the right hand sides of the equations and simplifying, until convergence is obtained.

As is usual, our implementation maintains a work-list of nodes whose postdominator set must be recomputed because one or more of the sets on the right hand side of its equation has changed. This work-list is initialized to $\texttt{END}_{\texttt{MAIN}}$, and nodes are enqueued and dequeued till convergence occurs. Postdominator sets are represented as bit-vectors, and the necessary union and intersection operations are performed using bit-vector operations.

3.2 Reachability-based Algorithm

Let us observe that, by a straightforward reformulation of Definition 6, a node v fails to postdominate precisely those nodes w for which there are valid path suffixes $w \rightarrow \texttt{END}_{\texttt{MAIN}}$ that do not contain v. This observation is the basis for our algorithm, the pseudocode for which is shown in Figure 3.

The algorithm determines the interprocedural postdominance relation `PDOM` by first initializing it to contain all pairs (v, w) (line 3) and then pruning from it, for each node v (processed by one iteration of the loop in line 7), all those nodes w that are reachable from $\texttt{END}_{\texttt{MAIN}}$ in the reverse ICFG with v deleted, via the reverse of a valid path suffix. The actual pruning is accomplished by the call $\texttt{SEARCH-WITHOUT}(v)$. This procedure (line 10) determines reachability by performing a search. A work-list of reachable nodes is maintained, which is initialized to contain $\texttt{END}_{\texttt{MAIN}}$ (line 13), and from which nodes are extracted one at the time (line 15). A node w is marked when it is first encountered (line 17), to avoid processing it multiple times, and the entry (v, w) in deleted

Call Nodes: $PDOM(\texttt{Call}_\texttt{F}) = PDOM(\texttt{START}_\texttt{F}) \cup PDOM(\texttt{Return}_\texttt{F}) \cup \{\texttt{Call}_\texttt{F}\}$

END_{MAIN}: $PDOM(\texttt{END}_\texttt{MAIN}) = \{\texttt{END}_\texttt{MAIN}\}$

Other Nodes: $PDOM(\texttt{n}) = \bigcap_{s \in succ(n)} PDOM(\texttt{s}) \cup \{\texttt{n}\}$

Figure 2: Rules for postdominance dataflow equations

from PDOM (line 19). Then, procedure VISIT is invoked (line 20) to add the predecessors of w to the work-list, where appropriate.

Reachability along valid path segments introduces some subtleties, specifically when $w = \texttt{START}_F$, for some procedure F. In this case, a generic predecessor c of w is a call node; hence a valid path suffix from w to $\texttt{END}_\texttt{MAIN}$ can be extended by prepending edge (c, w) if and only if such a path includes the corresponding return node $r(c)$. Care must be taken to handle correctly and efficiently the situation when $\texttt{START}_F$ is visited before $r(c)$ as well as that one the visit happens in the opposite order. To this end, when a return node $r(c)$ corresponding to a call node c for F is visited, in addition to adding $\texttt{END}_F$ to the work-list (line 25) node c is considered as well: if $\texttt{START}_F$ has already been visited, then c is added to the work-list (line 28), else c is inserted (line 29) into an initially empty *bucket* associated with procedure F. When $\texttt{START}_F$ is visited, the bucket - which contains call nodes whose corresponding return has already been visited - is simply emptied into the work-list (lines 30-35). For a node w that is neither a return nor a start node, the visit simply adds all predecessors to the work-list (lines 36-40).

We have assumed that each node is given a unique number between 1 and the total number N of nodes in the ICFG, and that corresponding call and return nodes can be determined from each other in constant time. Moreover, deletion of node v from the ICFG is not actually performed, but simulated simply by skipping the visit of v (line 18) and hence preventing the search to propagate beyond v.

Next, we analyze the performance and establish the correctness of our algorithm.

Theorem 1 *Given as input an ICFG $G = (V, E)$, procedure* COMPUTE-PDOM *runs in time $O(|V|(|V| + |E|))$ and sets* PDOM*[v][w] to true if and only if v interprocedurally postdominates w.*

Proof: The performance bound simply follows from the fact that $|V|$ searches are executed, each taking time proportional to the number $|V|$ of nodes and $|E|$ of edges.

Correctness requires a more detailed argument. We first consider separately the case $v = \texttt{END}_\texttt{MAIN}$, where v is added to the work-list in line 13 and is removed from the work-list in line 15. The while loop in line 14 executes just once and, for each w, the entry PDOM[$\texttt{END}_\texttt{MAIN}$][w] will remain true, which is correct since $\texttt{END}_\texttt{MAIN}$ postdominates all nodes.

Now, let $v \neq \texttt{END}_\texttt{MAIN}$. We will show that PDOM[v][w] will be set to false iff there is a non-empty valid path suffix from w to $\texttt{END}_\texttt{MAIN}$ that does not contain v.

Notationally, $y \to z$, $y \xrightarrow{*} z$, and $y \xrightarrow{+} z$ respectively denote an edge, a path, and a non empty path between nodes y and z.

- $\Leftarrow$: We show inductively that, for every n, if $\pi = w \xrightarrow{+} \texttt{END}_\texttt{MAIN}$ is a non-empty valid path suffix of length n that does not contain v, then PDOM[v][w] is set to false.

  If $n = 0$, then $w = \texttt{END}_\texttt{MAIN}$. This node is added to the work-list in line 13, removed from it in line 15, and PDOM[v][w] is set to false in line 19.

  Assume now that the inductive hypothesis holds for lengths no larger than n. Now consider a node w for which there is a non-empty valid path suffix $w \to x \xrightarrow{*} \texttt{END}_\texttt{MAIN}$ of length $n + 1$. By the inductive assumption, PDOM[v][x] will be set to false at some point. This must happen at line 19, which is immediately followed by line 20 which invokes procedure VISIT. Consider the three possible cases for node x.

  1. If node x is a return node and w is not marked, then w is added to the worklist in line 25. At some later point, it is removed from the worklist and PDOM[x][w] is set to false. On the other hand, if w is marked, it must have been marked in line 17 which is followed by line 19 in which PDOM[x][w] is set to false. In either case, the required result follows.
  2. If node x is the START node for some procedure F, w must be a call node for F and the corresponding return node $r(w)$ must occur on the valid path suffix $x \xrightarrow{*} \texttt{END}_\texttt{MAIN}$. By the inductive assumption, both x and $r(w)$ must be put on the work-list, and procedure VISIT must be called with both these nodes. If x is processed first, then line 28 is executed when $r(w)$ is processed at a later time, and w is put on the work-list. On the other hand, if $r(w)$ is processed first, then w is added to buckets[F] in line 29, and w is added to the worklist in line 34 when x is processed. In either case, at some point, w is extracted from the work list and PDOM[v][w] is set to false.
  3. The remaining case is when $w \to x$ is an internal edge. When x is processed, w is either already marked, in which case PDOM[v][w] is set to false, or w is not marked, in which case it is added to the worklist, and IPDOM[v][w] is set to false when w is processed.

- $\Rightarrow$: We note that elements of the PDOM array are set to false in the while loop of line 14 and we show inductively that, for each k, if PDOM[v][w] is set to false at the k-th iteration, then there is a valid path suffix π from w to $\texttt{END}_\texttt{MAIN}$ that does not contain v.

  The while loop executes at least once since node $\texttt{END}_\texttt{MAIN}$ is added to the worklist in line 13. PDOM[v][$\texttt{END}_\texttt{MAIN}$] is set to false

```
1.  ICFG G;
2.  int N = number of nodes in G;
3.  boolean[1..N][1..N] PDOM = true;
      // PDOM[v][w] = false if v does not postdominate w
4.  int P = number of procedures in G;
5.  node_set  work-list = {}; //set of nodes to be visited by graph search

6.  procedure COMPUTE-PDOM () {
7.    FOR v = 1..N DO
        // find nodes reachable from END-MAIN in reverse ICFG w/o v
8.      SEARCH-WITHOUT(v);
9.    END
    }

10. procedure SEARCH-WITHOUT(node v) {
11.   node_set[P] buckets = {};
12.   boolean[N] marked = false; // no node initially visited

13.   work-list.add(END-MAIN);
      // begin reverse reachability at END-MAIN
14.   WHILE(work-list <> empty) DO
15.      node w = work-list.remove();
16.      IF (marked[w]) THEN continue; //w has already been visited
17.      marked[w] = true;
18.      IF (w == v) continue; // skip v since it is conceptually removed from G
19.      PDOM[v][w] = false; // w is reachable from  END-MAIN w/o v
20.      VISIT(w, v, buckets, marked); // process predecessors of w
21.   END
    }

22. procedure VISIT(node w, node v, node_set[] buckets, boolean[] marked) {
      // process predecessors of w
23.   IF (w is a return node from procedure F) // w target of return edge
24.   THEN
25.       IF (not marked(END-F)) THEN work-list.add(END-F);
26.       ELSE
27.         let node c = c(w) // call node corresponding to w;
28.         IF (marked[START-F]) THEN work-list.add(c);
29.         ELSE buckets[F].add(c);

30.   ELSEIF (w is a START node for procedure F)  // w target of call edge
31.   THEN
32.      FOR each node c in buckets[F] DO
33.        remove c from buckets[F];
34.        work-list.add(c);
35.      END

36.   ELSE // w target of internal edge
37.       FOR each predecessor z of w DO
38.         IF (not marked[z]) THEN
39.           work-list.add(z);
40.       END
    }
```

Figure 3: Algorithm for computing the interprocedural postdominance relation

in the first iteration, and the required result is trivially obtained by letting π be the empty path, obviously a valid path suffix.

Assume now that the inductive assumption holds for the first k iterations of the while loop. Let w be the node removed from the work-list in iteration $k+1$ and assume $w \neq v$. Therefore, node w is distinct from $\mathtt{END_{MAIN}}$ and it must have been added to the work-list when some node r was processed in procedure VISIT. By the inductive assumption, there is a valid path

$$\pi = \mathtt{START_{MAIN}} \xrightarrow{*} r \xrightarrow{+} \mathtt{END_{MAIN}}$$

and the suffix $r \xrightarrow{+} \mathtt{END_{MAIN}}$ does not contain v. Consider the three possible cases for node r.

1. Suppose r is a return node for procedure F.
   Suppose $w = \mathtt{END}_F$. Since the only ICFG predecessor of node r is w, the valid path π must contain w, so there is a valid path suffix $w \xrightarrow{+} \mathtt{END_{MAIN}}$ that does not contain v.
   Otherwise, $w = c(r)$ is the call node corresponding to r. Then, any valid path that contains r must contain w since r so π contains w, and there is a valid path suffix
   $$\rho = w \rightarrow \mathtt{START}_F \xrightarrow{+} \mathtt{END}_F \xrightarrow{+} \mathtt{END_{MAIN}}.$$
   By the inductive assumption, there is a valid path suffix $\sigma = \mathtt{START}_F \xrightarrow{+} \mathtt{END}_F \xrightarrow{+} \mathtt{END_{MAIN}}$ that does not contain v. Path σ has a same-level valid path segment $\gamma = \mathtt{START}_F \xrightarrow{+} \mathtt{END}_F$. Replacing the same-level valid path segment from $\mathtt{START}_F$ to $\mathtt{END}_F$ in ρ with γ, we get a valid path suffix $w \xrightarrow{+} \mathtt{END_{MAIN}}$ that does not contain v.
2. Suppose r is a `START` node for procedure F.
   Then w must be a call node for F which was in `buckets`$[F]$. This node must have been added to `buckets`$[F]$ in line 29 when the return node r' corresponding to w was processed by procedure `VISIT`. Node r' must have been processed before node r, so by inductive assumption, there is a valid path $\mathtt{START_{MAIN}} \xrightarrow{*} r' \xrightarrow{+} \mathtt{END_{MAIN}}$ which must contain w, so the required result holds.
3. The final case is that $w \rightarrow r$ is an internal edge.
   By assumption, there is a valid path $\pi = \mathtt{START_{MAIN}} \xrightarrow{*} r \xrightarrow{+} \mathtt{END_{MAIN}}$. Let F be the procedure in which r occurs. By assumption, $\mathtt{START}_F$ must occur on π, which can then be written as
   $$\pi = \mathtt{START_{MAIN}} \xrightarrow{*} \mathtt{START}_F \xrightarrow{*} r \xrightarrow{+} \mathtt{END_{MAIN}}$$
   By assumption about programs, there is a same-level valid path segment $\beta = \mathtt{START}_F \xrightarrow{*} q \rightarrow r$. Replacing the segment $\mathtt{START}_F \xrightarrow{*} r$ with β, we get a valid path that contains w as required; furthermore, the suffix $w \xrightarrow{+} \mathtt{END_{MAIN}}$ does not contain v.

□

4 Precomputing Reachability

We now show that the efficiency of the algorithm for computing the interprocedural postdominance relation can be improved by precomputing and caching *intraprocedural* reachability information, and using this information selectively to ensure that most of the graph traversals performed during the postdominance computation are along interprocedural edges.

In the preprocessing step, for each procedure $P \in \mathcal{P}$, we perform reachability computations in the intraprocedural control flow subgraph $S_P = (V_P, E_P^0)$, where the shortcut edges have been removed. The nodes in $V_P^c \cup \{\mathtt{END}_P\}$ (namely, the call nodes and $\mathtt{END}_P$) are called *root nodes* of this graph. We introduce the *locally reachable node set* $L(r)$ which is the set of nodes in V_P' (nodes that are not $\mathtt{START}_P$, $\mathtt{END}_P$ or call or return nodes, as in Definition 1) reachable from root node r in the reverse graph of S_P. Intuitively, in the reverse ICFG, nodes in $L(r)$ can be reached from the r without traversing interprocedural edges. We then build a collapsed graph $C_P = (I_P, D_P)$ in which the nodes are $\mathtt{START}_P$, $\mathtt{END}_P$, and all the call and return nodes of P, and in which there is an edge (m, n) if $m, n \in I_P$ and $m \in L(n)$. At the $\mathtt{END}_P$ node and each call node in this collapsed graph, we store the locally reachable set of nodes computed for the corresponding node in S_P (the algorithm below assumes that this set is stored in an array named `L`, indexed by the node). During the postdominator computation, we use S_P for reachability computations if the deleted node is not in P; otherwise, we use the original ICFG for P.

Figure 4 shows the collapsed graphs for the procedures in the running example. Figure 5 shows the modifications that are required to the algorithm of Figure 3, when preprocessing is used.

The proof of correctness of the algorithm with preprocessing hinges on the fact that the intraprocedural reachability computation that determines locally reachable sets finds same-level paths within procedures which, prepended to paths from the `END` nodes of the procedure to $\mathtt{END_{MAIN}}$, yield valid path suffixes.

5 Computation of Transitive Reduction

`PDOM` is a reflexive, anti-symmetric, transitively closed relation. Boolean array `PDOM`[v][w] can be viewed as the adjacency matrix of a graph where there is an edge $(u \rightarrow v)$ in the graph if u postdominates v. This graph has self-loops at every node (reflexivity); except for self-loops, it is acyclic (anti-symmetricity), and, if $(u \rightarrow v)$ and $(v \rightarrow w)$ are edges in the graph, so is edge $(u \rightarrow w)$ (transitive closure).

For some applications, it is useful to work the transitive reduction `IPDOM`, which can be computed by a single boolean matrix multiplication, as we show in this section.

The first step is to define the irreflexive version P_I of `PDOM`, by the equation

$$P_I = PDOM \wedge \neg I_n, \tag{1}$$

where I_n is the $n \times n$ identity matrix, whose diagonal entries are true and the other are false. We show next that the transitive reduction $IPDOM$ of `PDOM` can be computed by the following expression:

$$IPDOM = P_I \wedge \neg P_I^2. \tag{2}$$

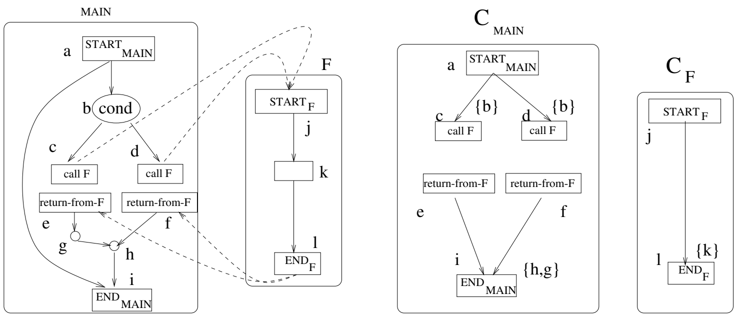


Figure 4: Collapsed Graphs for Running Example

```
// Assume that CF is the collapsed graph for procedure F
......
19a.  FOR each node n in L[w] DO
  b.      PDOM[v][n] = false;
......
22. procedure VISIT(node r, node del, node_set[] buckets, boolean[] marked) {
     // process predecessors of w
23.  IF (r is a return node from procedure F)
24a. THEN
  b.     IF (v is in F) THEN GRAPH-F = SF
  c.     ELSE GRAPH-F = CF;
25.      IF (not marked(END-GRAPH-F)) THEN work-list.add(END-GRAPH-F);
26.      ELSE
27.        let node c = c(w) \\ call node corresponding to w;
28.        IF (marked[START-GRAPH-F]) THEN work-list.add(c);
29.        ELSE buckets[F].add(c);
.......
    }
```

Figure 5: Postdominator Computation with Preprocessing

Lemma 2 *The relation $IPDOM$ is transitively reduced.*

Proof: Note first that if $(i,k) \in IPDOM$, then $(i,k) \in P_I$ (from Equation 2), so $(i,k) \in P$ and i is distinct from k (from Equation 1).

Suppose $(i,j),(j,k) \in IPDOM$. We have just shown that $(i,j),(j,k) \in P_I$, so $(i,k) \in P_I^2$. Therefore, $(i,k) \notin IPDOM$. □

Theorem 2 $IPDOM^* =$ PDOM.

Proof:

- $IPDOM^* \subseteq$ PDOM: We show that $IPDOM \subseteq$ PDOM; since PDOM is transitively closed, this proves the required result. From the definitions of $IPDOM$ and P_I, we obtain the following equation:

  $IPDOM = P_I \wedge \neg P_I^2 = \mathtt{PDOM} \wedge \neg I_n \wedge P_I^2$

  The conjunction of PDOM with other matrices never has more $true$ entries than PDOM itself, so $IPDOM \subseteq$ PDOM.

- PDOM $\subseteq IPDOM^*$: If $(x,y) \in$ PDOM, let

  $$x = z_0, z_1, z_2, ..., z_n = y$$

  be the longest path from x to y in the graph of PDOM where $z_0, z_1, ..., z_n$ are all distinct nodes (such a path exists because the graph, without self-loops, is acyclic). Let us call this path R. For all k, z_k and z_{k+1} are distinct nodes, so $(z_k, z_{k+1}) \in P_I$; furthermore, $(z_k, z_{k+1}) \notin P_I^2$ since otherwise, there is a path of length 2 from z_k to z_{k+1} in the graph of P_I, and hence in the graph of PDOM, contradicting the assumption that R is the longest path from x to y in the graph of PDOM.

□

These facts lead to the following result.

Lemma 3 *The transitive reduction of the interprocedural postdominance relation can be computed in time $O(|V| * (|E| + |V|^2))$.*

6 Computation of Interprocedural Control Dependence

In this section, we briefly outline how to compute the control dependence relation C and its related sets (Definitions 8 and 9). One can represent C as an $|E| \times |V|$ Boolean array whose entry $C[e,w]$ is set to true iff node w is control dependent on edge e. Alternatively, C can be considered as the edge set of a bipartite graph $B = (E, V, C)$, with vertex sets E and V. Both representations have size $O(|V||E|)$, in the worst case; if the relation is sparse, however, the size of B could be much smaller.

Relation C can be easily constructed from the interprocedural postdominance relation by simply scanning all pairs $(e = (u,v), w)$ and checking the entries PDOM[w][u] and PDOM[w][v]. This takes time $O(|V||E|)$, for both the array and the graph representations.

In the graph representation, set $\mathtt{cd}(e)$ is easily obtained, in time proportional to its size, by collecting the neighbors of e in B. A similar approach can be used to obtain $\mathtt{conds}(w)$.

Finally, the `cdequiv` sets, which are the classes of equivalence of nodes that have the same dependences, can be obtained by the following approach. A partition of V, initially consisting of just one set, is progressively refined until, at the end, the blocks of the partition coincide with the `cdequiv` sets. A refinement phase is performed for each edge e, by splitting each block of the partition into two subblocks respectively containing the nodes that are and are not control dependent on e. It is easy to see that the entire process can be completed in time $O(|V||E|)$.

7 Experimental Results

We have implemented our algorithm using the SUIF1 C front end [32] and the MachSUIF control flow graph library [16]. On top of the machsuif code, we have written the code to bind the procedure control flow graphs into an interprocedural control flow graph. The implementation of our algorithm with preprocessing requires only about 250 lines of C++ code once the supporting data structures have been created.

The data in Table 1 compares the performance of our algorithm with preprocessing with that of the aforementioned worklist based iterative algorithm on five sample codes.

- mutual: A test program involving mutual recursion
- postfix: A postfix calculator
- llist: A linked list class and driver
- schedule: A simple priority scheduler
- eqntott: Logical expression to truth table translator

The column $PDOM$ shows the number of non-zero entries in the full postdominance relation. The columns *Iter Set* and *Iter List* indicate the number of set and worklist operations performed by the iterative algorithm, respectively. The columns *Reach Set* and *Reach List* indicate the number of set and worklist operations performed by our algorithm, respectively.

Since our algorithm makes only local updates to the PDOM relation when a node is processed we have been able to greatly reduce the number of set update operations performed. This is in contrast to the iterative technique which must perform union and intersection operations over bit vectors of length equal to the number of nodes in the program at each update. By storing the results of our preprocessing searches in small, single procedure sized, bit vectors we are able to get most of the advantage of updating a word length bit vector segment in a single operation without the added expense of updating the entire bit vector.

Since our algorithm relies upon repeated searches, we generally require more constant time list operations. However, in practice these operations are only slightly more expensive than the set operations and so the net result is a significant performance improvement.

Code	Nodes	Edges	Procs	Calls	PDOM	Iter Set	Iter List	Reach Set	Reach List
mutual	23	29	4	6	136	2691	108	583	148
postfix	88	135	6	24	1094	98120	1123	14197	2068
llist	201	283	13	33	6928	320597	1650	52843	7452
schedule	343	463	19	27	3526	821115	2626	49293	5205
eqntott	1463	2147	55	114	9600	12866247	9663	1060671	52832

Table 1: Algorithm Performance Results

8 Conclusions and Future Work

In this paper we have presented an efficient algorithm for computing interprocedural control dependence by first computing interprocedural postdominance. While the algorithm we present runs in time quadratic in the size of the CFG, it requires that the entire postdominance relation be computed. Our future goal is to develop an algorithm to compute the postdominance DAG directly in a manner similar to the one by which Lengauer and Tarjan [19] directly compute the postdominance tree in the intraprocedural case. This would eliminate the need to compute a transitive reduction as well as ameliorate the space demands of present algorithms which often make them infeasible for very large programs.

From the DAG representation of the postdominance relation our hope is to extend the Roman chariots [24] formulation of control dependence queries to this case. Such an extension would allow us to answer control dependence queries more quickly and without the need to either store or explicitly compute the entire control dependence relation.

It is also our goal to incorporate into our program model atypical control flow effects such as embedded halts and exception handling. Recent work [31] suggests that such effects can be incorporated by augmenting the existing program representation with additional nodes and edges, and then acting in some appropriate way when traversing the additional edges to preserve context. It is our belief that the core algorithm presented here will extend naturally to those representations.

Finally, we hope to integrate these algorithms into a toolkit to do both interprocedural control and data flow analysis as well as program slicing analysis on large programs.

9 Acknowledgments

The work of G. Bilardi was supported in part by MURST of Italy and by IBM while this author was a visitor at the T. J. Watson Research Laboratory.

References

[1] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for determining useful parallelism. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 207–215, St. Malo, France, July 4–8, 1988.

[2] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255, Toronto, Ontario, June 26–28, 1991.

[3] G. Bilardi and K. Pingali. A framework for generalized control dependence. In *Proceedings of the ACM SIGPLAN '96 conference on Programming language design and implementation*, pages 291–300, May 1996.

[4] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, May 1998.

[5] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] R. Cytron, J. Ferrante, and V. Sarkar. Compact representations for control dependence. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 337–351, White Plains, New York, June 20–22, 1990.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] J. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 7(3):478–490, 1981.

[10] R. Gupta, L. Pollock, and M. L. Soffa. Parallelizing data flow analysis. In *Proceedings of the Workshop*

on Parallel Compilation, Kingston, Ontario, May 6–8, 1990. Queen's University.

[11] R. Gupta and M. L. Soffa. Region scheduling. In *2nd International Conference on Supercomputing*, pages 141–148, 1987.

[12] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. Technical Report OSU-CISRC-7/97-TR36, The Ohio State University, July 1997.

[13] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Software Engineering Notes, Proceedings of the International Symposium on Software Testing and Analysis*, volume 23, pages 11–20. ACM/SIGSOFT, March 1998.

[14] S. Horowitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 133–145, Munich, West Germany, Jan. 21–23, 1987.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Transactions on Programming Languages and Systems*, 12(1):22–60, January 1990.

[16] HUBE Compiler Group. The MachSUIF library. The MachSUIF documentation set, 1997. Harvard University.

[17] R. Johnson. *Efficient Program Analysis using Dependence Flow Graphs*. PhD thesis, Cornell University, August 1994.

[18] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, Orlando, Florida, June 20–24, 1994.

[19] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[20] D. Liang and M. J. Harrold. Slicing objects using system dependence graph. In *Internation Conference on Software Maintenance*, November 1998.

[21] J. Loyall and S. Mathisen. Using dependence analysis to support the software maintenance process. In *Proceedings of the Conference on Software Maintenance*, pages 282–291, September 1993.

[22] C. Newburn, D. Noonburg, and J. Shen. A PDG-based tool and its use in analyzing program control dependences. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, August 1994.

[23] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, April 1984.

[24] K. Pingali and G. Bilardi. Optimal control dependence and the roman chariots problem. *ACM Transactions on Programming Languages and Systems*, 19(3):1–30, May 1997.

[25] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–79, September 1990.

[26] L. Pollock and M. Soffa. An incremental version of iterative dataflow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, December 1989.

[27] B. G. Ryder and M. C. Paull. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[28] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural data flow analysis for with applications to constant propogation. Technical Report TR-1284, Computer Science Deparment, University of Wisconsin, Madison, August 1995.

[29] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[30] B. Simons, D. Alpern, and J. Ferrante. A foundation for sequentializing parallel code. In *Conference Record of SPAA '90: ACM Symposium on Parallel Algorithms and Architecture*, July 1990.

[31] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *International Conference on Software Engineeting*, pages 432–441, May 1999.

[32] Stanford Compiler Group. The SUIF library. The SUIF compiler documentation set, 1994. Stanford University.