A Framework for Generalized Control Dependence

Gianfranco Bilardi
DEI, Università di Padova, 35131 Padova, Italy
EECS, University of Illinois, Chicago, IL 60607

Keshav Pingali
Department of Computer Science
Cornell University, Ithaca, NY 14853

Abstract

We generalize the notion of *dominance* by defining a generalized dominance relation with respect to a set of paths in the control flow graph $G = (V, E)$. This new definition leads to a generalized notion of *control dependence*, which includes *standard control dependence* and *weak control dependence* as special cases.

If the set of paths underlying a generalized dominance relation satisfies some natural closure conditions, that dominance relation is tree-structured. Given this tree, the corresponding control dependence relation can be computed optimally by reduction to the *Roman Chariots Problem*, which we have developed previously for computing standard control dependence. More precisely, given linear preprocessing time and space, we can answer the (generalized version of the) so called `cd`, `conds`, and `cdequiv` queries in time proportional to the output of the query.

To illustrate the utility of the framework, we show how weak control dependence can be computed optimally in $O(|E|)$ preprocessing space and time. This improves the $O(|V|^3)$ time required by the best previous algorithm for this problem.

1 Introduction

Control dependence was introduced by Ferrante, Ottenstein and Warren [FOW87] to solve a number of problems in program analysis and parallelization. In this paper, we refer to it as *classical control dependence* since it was the first formalization of the intuitive idea that the execution of predicate nodes in a program determines how control flows through a program. For some applications, other definitions of control dependence have proved to be useful. For example, Podgurski and Clarke have proposed the notion of *weak control dependence* for proving total correctness of programs [PC90]. Both classical and weak control dependence are reviewed in Section 2.

In Section 3, we present a generalized notion of control dependence, and show that classical and weak control dependence are special cases of this general definition. We do this as follows. First, we define a generalized notion of dominance which is parameterized with respect to a set of paths in the control flow graph (CFG). Then, we use this to generalize the notion of control dependence in a natural way, and show that classical and weak control dependence are special cases of this general notion of control dependence. Applications of control dependence usually require answers to queries known as `cd`, `conds`, and `cdequiv` in the literature [CFS90]. For classical control dependence, we have developed an optimal computational approach based on a reduction to the *Roman Chariots Problem* [PB95]. For generalized control dependence, we show that if the set of paths used in the underlying dominance relation satisfies some natural closure properties, the dominance relation is tree-structured; for such control dependence relations, our solution to the Roman Chariots Problem immediately provides an optimal computational approach.

In Sections 4 and 5, we focus on weak control dependence, and its underlying dominance relation, which we call *loop postdominance*. In Section 4, we show that the loop postdominance relation is tree structured, and that its transitive reduction, the loop postdominance forest, is a pruning of the standard postdominator tree. This fact was proved earlier by Podgurski in his the-

[1]Gianfranco Bilardi (bilardi@art.dei.unipd.it) was supported in part by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM) and by the Italian Ministry of University and Research.

[2]Keshav Pingali (pingali@cs.cornell.edu) was supported by NSF grants CCR-9008526 and CCR-9503199, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

sis [Pod89]. However, we give a new characterization of the set of edges to be pruned, in terms of a set of nodes called *crowns*. In Section 5, we give an $O(|E|)$ algorithm based on depth-first search for computing the set of crowns. Crowns are used to build the loop postdominator forest in $O(|E|)$ time. Using this data structure, we exploit our solution to the Roman Chariots Problem to solve the problem of answering `cd`, `conds` and `cdequiv` queries on the weak control dependence relation optimally (that is, in $O(|E|)$ preprocessing time, and query time proportional to the size of the answer to a query). This improves the $O(|V|^3)$ preprocessing time required by Podgurski and Clarke's algorithm for answering `cd` and `conds` queries; it also provides an optimal algorithm for answering `cdequiv` queries.

2 Two Control Dependence Relations

The following definitions are standard.

Definition 1 *A* **control flow graph (CFG)** $G = (V, E)$ *is a directed graph in which nodes represent statements, and an edge $u \rightarrow v$ represents possible flow of control from u to v. Set V contains two distinguished nodes:* START, *with no predecessors and from which every node is reachable; and* END, *with no successors and reachable from every node.*

It is convenient to assume that E contains edges START $\rightarrow$ END (as in [FOW87]) and END $\rightarrow$ END.

Definition 2 *A node w* **postdominates** *a node v if every path from v to* END *contains w. If, in addition, $w \neq v$, then w is said to* **strictly postdominate** *v.*

It can be shown that postdominance is a transitive relation with a tree-structured transitive reduction called the *postdominator tree*, which can be constructed in $O(|E|\alpha(|E|))$ time by an algorithm due to Tarjan and Lengauer [LT79], or in $O(|E|)$ time by a rather more complicated algorithm due to Harel [Har85].

Classical control dependence can be defined formally as follows [FOW87].

Definition 3 *A node w is* **control dependent** *on edge $(u \rightarrow v) \in E$ if*

1. *w postdominates v, and*
2. *w does not strictly postdominate u.*

Intuitively, this means that if control flows from node u to node v along edge $u \rightarrow v$, 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. Figure 1 shows a CFG, its postdominator tree and its classical control dependence relation. We will often use the term control dependence, without any qualifications, to refer to classical control dependence.

Podgurski and Clarke have introduced *weak control dependence* [PC90], which is more appropriate than the classical one for proving total correctness of programs. In this paper, we call it *loop control dependence* because we give an alternative definition of it based on the concept of *loop postdominance*, given below. Figure 1 shows a program in which classical and loop control dependence differ. Consider node k in the CFG. In the classical notion, k is control dependent on $g \rightarrow a$. However, to prove that k will be executed, it is necessary to prove that the loop $a \rightarrow b \rightarrow a$ terminates. Therefore, in the context of proving total correctness of programs, it is more appropriate to make k loop control dependent on the exit of the loop — namely, the edge $b \rightarrow k$. In programs without cycles, the classical and loop control dependence relations are identical.

Definition 4 *Assume that* END $\rightarrow$ END $\in E$. *A node w* **loop postdominates** *a node v if every infinite path starting at v contains w. If, in addition, $w \neq v$, then w is said to* **strictly loop postdominate** *v.*

In other words, if control reaches a node v, and w loop postdominates v, then control will reach w in a finite number of steps, whether or not the program terminates (that is, whether or not control reaches END).

Definition 5 *A node w is* **loop control dependent** *on edge $(u \rightarrow v) \in E$ if*

1. *w loop postdominates v, and*
2. *w does not strictly loop postdominate u.*

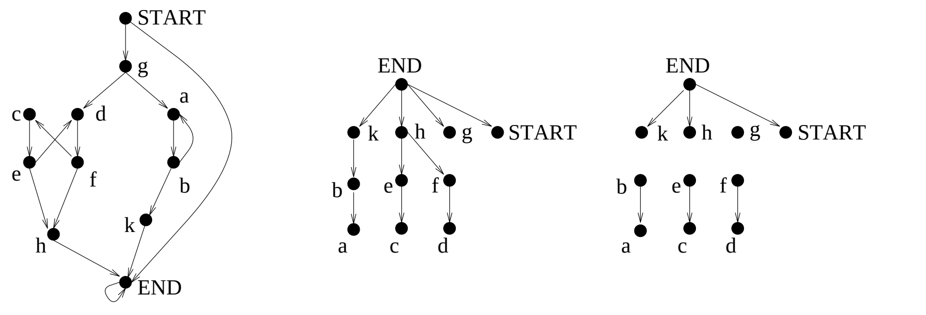
Intuitively, this means that (i) if control reaches v, it must reach w in a finite number of steps, and (ii) from u, it is possible for control to reach a cycle of nodes (possibly, the self loop at END) without encountering w. It can be shown that loop control dependence is equivalent to Podgurski and Clarke's weak control dependence.

In applications of any control dependence relation, the following queries arise naturally for a given edge e or node v [CFS90]:

1. `cd`(e): which nodes are control dependent on e?
2. `conds`(v): which edges is v control dependent on?
3. `cdequiv`(v): which other nodes are control dependent on the same set of edges as v?

3 The Framework

We now discuss a general framework which unifies classical and weak control dependence, and supports the design of optimal algorithms for answering queries on such relations.



(a) Control Flow Graph (b) Postdominator tree (c) Loop postdominator forest

E \ V	a	b	c	d	e	f	g	h	k
START $\rightarrow$ g							● □		
g $\rightarrow$ d				● □		● □		●	
g $\rightarrow$ a	● □	● □							●
f $\rightarrow$ c			● □		● □				
e $\rightarrow$ d				● □		● □			
b $\rightarrow$ a	● □	● □							
e $\rightarrow$ h								□	
f $\rightarrow$ h								□	
b $\rightarrow$ k									□

● : control dependence

□ : loop control dependence

(d) Control Dependence and Loop control dependence

Figure 1: A Program and Two Control Dependence Relations

By the standard definition of postdominance, w postdominates v if every path from v to `END` contains w. In other words, all terminating executions that reach v eventually reach w. We generalize this notion by considering an arbitrary class of possible executions, modeled as a set of paths $\mathcal{P}$ in the CFG. Different notions of dominance and, correspondingly, of control dependence, arise from different choices for $\mathcal{P}$.

Let I be a (possibly infinite) interval on the integer line. A path of the CFG is a sequence of nodes $\{v_i \in V : i \in I\}$ such that if $i, i+1 \in I$ then $(v_i \rightarrow v_{i+1}) \in E$. A path is *finite, of length* n, when $I = \{0, 1, \ldots, n\}$ and becomes *trivial* when $n = 0$. A path is *right infinite* when I is the set of positive integers and is *left infinite* when I is the set of negative integers.

Definition 6 *Given a set of paths $\mathcal{P}$ in a directed graph $G = (V, E)$, let $\mathcal{P}_v$ denote the subset of paths in $\mathcal{P}$ that contain $v \in V$. We say that v $\mathcal{P}$-***dominates** *u, denoted $v \vdash u$, if and only if $\mathcal{P}_v \supseteq \mathcal{P}_u$.*

Technically, $\vdash_{\mathcal{P}}$ would be a more appropriate notation; we omit the subscript for simplicity.

Definition 7 *We say that $w \in V$ is $\mathcal{P}$-***control dependent** *upon edge $(u \rightarrow v) \in E$ if*

- *$w \vdash v$, and*
- *if $u \neq w$, then $w \not\vdash u$.*

Here are some interesting path sets, and the corresponding dominance relations.

Definition 8

1. *$\mathcal{S}$: the set of finite paths starting at* `START`*.*
   $\mathcal{S}$-dominance = predominance (classical dominance).
2. *$\mathcal{E}$: the set of finite paths ending at END.*
   $\mathcal{E}$-dominance = postdominance.
3. *$\mathcal{B}$: the set of finite paths from* `START` *to* `END`*.*
   $\mathcal{B}$-dominance = predominance OR postdominance.
4. *$\mathcal{L}$: the set of left-infinite paths.*
   $\mathcal{L}$-dominance = loop predominance.
5. *$\mathcal{R}$: the set of right-infinite paths.*
   $\mathcal{R}$-dominance = loop postdominance.

In the following two subsections, we study properties of $\mathcal{P}$ that lead to a forest-structured dominance relation and enable the formulation of control dependence queries in terms of the Roman Chariots Problem.

3.1 The $\mathcal{P}$-dominance Relation

The following properties of a set of paths $\mathcal{P}$ are of interest:

Definition 9

- Prefix Closure*: Any prefix of a path in $\mathcal{P}$ is in $\mathcal{P}$.*
- Suffix Closure*: Any suffix of a path in $\mathcal{P}$ is in $\mathcal{P}$.*
- Junction Closure*: Whenever $\pi_1 v \pi_2$ and $\sigma_1 v \sigma_2$ are in $\mathcal{P}$, then $\pi_1 v \sigma_2$ is also in $\mathcal{P}$.*
- Preaugmentation Closure*: If $(u \rightarrow v) \in E$ and $(v\pi) \in \mathcal{P}$ then $(uv\pi) \in \mathcal{P}$.*

- Postaugmentation Closure*: If* $(u \rightarrow v) \in E$ *and* $(\pi u) \in \mathcal{P}$ *then* $(\pi u v) \in \mathcal{P}$*.*

It is easy to verify that $\mathcal{S}$ and $\mathcal{L}$ satisfy prefix, postaugmentation, and junction closure, $\mathcal{E}$ and $\mathcal{R}$ satisfy suffix, preaugmentation, and junction closure, while $\mathcal{B}$ satisfies junction closure. It is trivial to show that suffix and preaugmentation closure (or symmetrically, prefix and postaugmentation closure) of $\mathcal{P}$ imply junction closure of $\mathcal{P}$.

Next, we establish sufficient conditions on $\mathcal{P}$ that guarantee that the $\mathcal{P}$-dominance relation is forest-structured.

Theorem 1 *Let $\mathcal{P}$ be a set of paths with prefix (or suffix) and junction closure, such that set $\mathcal{P}_v$'s are not empty and are distinct from each other.*

Then, $\mathcal{P}$-dominance is a partial order *(i.e., a reflexive, transitive, and antisymmetric relation), and its transitive reduction is* forest-structured *(i.e., a node has at most one predecessor in the reduction).*

Proof: Reflexivity and transitivity follow from Definition 6 and the analogous properties of set inclusion. Antisymmetry follows from the distinctness of the $\mathcal{P}_v$'s.

Below, we assume suffix closure, the argument for prefix closure being similar.

We begin by claiming that ($v \vdash u$ and $w \vdash u$) implies ($v \vdash w$ or $w \vdash v$). In fact, let $\pi \in \mathcal{P}_u$ (exploiting the assumption that $\mathcal{P}_u$ is not empty). By definition of dominance, π contains both v and w. By the suffix property, assume that π starts at u and, w.l.o.g., assume that v occurs before w on π. Then, we can write $\pi = \pi_1 v \pi_2$, with π_1 w-free. Let now $\sigma = \sigma_1 v \sigma_2$ be any path in $\mathcal{P}_v$. By the junction property, $\pi_1 v \sigma_2 \in \mathcal{P}_u$, hence it contains w. Since π_1 is w-free, w must occur on σ_2. Therefore, $\sigma \in \mathcal{P}_w$. In conclusion, $\mathcal{P}_v \subseteq \mathcal{P}_w$, hence $w \vdash v$.

Assume that (u, v) and (u, w) are related by the transitive reduction of $\mathcal{P}$-dominance. We claim that $v = w$. In fact, by the previous claim, either v dominates w or vice versa. Say $v \vdash w$. Then, $u \vdash w$ is a transitive consequence of $u \vdash v$ and $v \vdash w$; therefore the pair (u, w) can not be in the transitive reduction, unless $w = v$. In conclusion, at most one node can immediately $\mathcal{P}$-dominate u, which implies that the $\mathcal{P}$-dominance relation is forest-structured. □

Theorem 1 and the observations preceding it immediately lead to the following result.

Corollary 1 *The transitive reductions of the classical dominance relation, classical postdominance relation, loop predominance relation, and loop postdominance relation are all forest-structured.*

It is useful to represent the transitive reduction of a $\mathcal{P}$-dominance relation as a graph, called the $\mathcal{P}$-*dominance graph.* Under the assumptions of Theorem 1, this graph is a forest, directed from root to leaves. It is convenient to make this forest into a tree by adding to V a distinguished node ∞ and an edge from it to each node of V with no parent in the forest (such as nodes g,b,e and f in Figure 1). Node ∞ is not shown in Figure 1(c) to avoid cluttering the diagram.

The tree just introduced will be referred to as the $\mathcal{P}$-**dominator tree**. In it, each node $v \in V$ has a parent called the *immediate $\mathcal{P}$-dominator* of v, and denoted $i\mathcal{P}dom(v)$. When $\mathcal{P} = \mathcal{S}$, the $\mathcal{P}$-dominator tree becomes the classical dominator tree, and $i\mathcal{P}dom(v)$ is the immediate dominator of v. Similarly, when $\mathcal{P} = \mathcal{E}$, the classical postdominator tree is obtained and $i\mathcal{P}dom(v)$ is the immediate postdominator of v.

3.2 Control Dependence Computations and the Roman Chariots Problem

In [PB95], it was shown that the computation of classical control dependence can be reduced to the *Roman Chariots Problem.* Based on the following result, we extend this reduction to any forest-structured $\mathcal{P}$-dominance relation.

Theorem 2 *Given a $\mathcal{P}$-dominator tree and a control flow edge $(u \rightarrow v) \in E$, let z denote the least common ancestor (denoted by lca) of v and $i\mathcal{P}dom(u)$. Then, the nodes of V that are $\mathcal{P}$-control dependent upon $(u \rightarrow v)$ are exactly those in the simple path in the $\mathcal{P}$-dominator tree from v to z, not including z.*

Proof: Consider a node w that is $\mathcal{P}$-control dependent upon $(u \rightarrow v)$. With reference to the two clauses of Definition 6, we have:

- w must $\mathcal{P}$-dominate v hence, by the tree-structured property, must lie on the path σ_v from v to the root ∞ of the $\mathcal{P}$-dominator tree;
- if $w \neq u$, then w must not $\mathcal{P}$-dominate u hence, by the tree-structured property, must lie outside the path $\sigma_{i\mathcal{P}dom(u)}$ from $i\mathcal{P}dom(u)$ to the root ∞ of the $\mathcal{P}$-dominator tree.

Then, w must lie on $\sigma_v - \sigma_{i\mathcal{P}dom(u)}$. This difference is a path starting at v and going up to, but not including the first intersection of σ_v and $\sigma_{i\mathcal{P}dom(u)}$, which is the nearest common ancestor z of v and $i\mathcal{P}dom(u)$. □

For example, in Figure 1, the nodes that are loop control dependent on $(g \rightarrow d)$ are d and f. The immediate loop postdominator of g is ∞, and the least common ancestor of d and ∞ is ∞. The nodes that are loop control dependent on $(g \rightarrow d)$ are the nodes on the path from d to ∞, excluding ∞ — namely, nodes d and f. Given the $\mathcal{P}$-dominator tree, this least common ancestor computation for *all CFG* edges simultaneously can be done in $O(|E|)$ time using the well-known algorithm of Harel and Tarjan [HT84].

We now recall the formulation of the Roman Chariots problem.

Roman Chariots Problem: *The major arteries of the Roman road system are organized as a rooted tree in which nodes represent cities, edges represent roads and the root represents Rome. Public transportation is provided by chariots, and the cities on each chariot route are totally ordered by the ancestor relation in the tree. Given a rooted tree $T =< V, F, ROME >$ and an array $A[1..m]$ of chariot routes in which each route is specified by its end points, design a data structure to answer the following queries optimally.*

1. `cd`(e): *Enumerate the cities on route e.*
2. `conds`(v): *Enumerate the routes that serve city v.*
3. `cdequiv`(v): *Enumerate the cities served by exactly the same routes that serve city v.*

To make the connection with the control dependence problem, let the $\mathcal{P}$-dominator tree be the rooted tree T. For each edge $(u \to v) \in E$, insert in array A a chariot route with endpoints v and $lca(v, i\mathcal{P}dom(u))$, as described in Theorem 2. Then, a control dependence query can be reformulated immediately as a Roman Chariots query. Therefore, these queries can be processed in time proportional to their output size, using the $\mathcal{APT}$ (augmented $\mathcal{P}$-dominator tree) data structure introduced in [PB95].

Therefore, given a CFG $G = (V, E)$ and a set of paths $\mathcal{P}$ satisfying the assumptions of Theorem 1, we can build a data structure to answer $\mathcal{P}$-control dependence queries optimally, *provided we can build the $\mathcal{P}$-dominance tree efficiently.*

3.3 An Important Special Case

For classical control dependence, it is well-known that the nodes that are control dependent on a given edge form a simple path in the postdominator tree [FOW87]. Indeed, in this special case, a stronger property holds: if $(u \to v) \in E$, then $i\mathcal{P}dom(u)$ is an ancestor of v. Therefore, we see that the least common ancestor of v and $i\mathcal{P}dom(u)$ is $z = i\mathcal{P}dom(u)$, and this node can be identified without recourse to the Harel and Tarjan algorithm.

Although not essential, this property does simplify the reduction to the Roman Chariot Problem, and it can be useful in other ways. We show below that suffix and preaugmentation closure are sufficient conditions for it. (As these two properties together imply junction closure, Theorem 1 holds.)

Proposition 1 *Let $\mathcal{P}$ be a set of paths with suffix and preaugmentation closure. If $(u \to v) \in E$, then every $w \neq u$ that $\mathcal{P}$-dominates u also $\mathcal{P}$-dominates v, i.e., $i\mathcal{P}dom(u)$ is an ancestor of v.*

Proof: Let $\pi \in \mathcal{P}_v$. Then, π contains v and hence a suffix of the form $v\sigma$. Due to suffix closure, $(v\sigma) \in \mathcal{P}_v$. Due to the preaugmentation property, we have that $(uv\sigma) \in \mathcal{P}$. As u occurs on this path, any w that $\mathcal{P}$-dominates u must also occur on it, in particular (being $w \neq u$) w must occur on the portion $v\sigma$. Since the latter is a suffix of π, we conclude the w occurs on π, and that $\mathcal{P}_v \subseteq \mathcal{P}_w$. □

An analogous proposition holds under prefix and postaugmentation closure. The assumptions of Proposition 1 simplify the algorithm considerably. If these assumptions are met, then the set of nodes control dependent on an edge $u \to v$ are simply the nodes on the simple path from v to $parent(u)$, excluding $parent(u)$.

The critical step remains the computation of the $\mathcal{P}$-dominator tree. Linear time solutions are known [Har85] for $\mathcal{P} = \mathcal{S}$ and $\mathcal{P} = \mathcal{E}$, that is, for the classical dominator and postdominator tree. In the rest of the paper, we develop a linear time algorithm for loop postdominance ($\mathcal{P} = \mathcal{R}$).

4 Relating Loop Postdominance to Postdominance

We now take a close look at the relation between $\mathcal{R}$-dominance, also called loop postdominance, and $\mathcal{E}$-dominance, i.e., classical postdominance. We shall assume that the edge `END` $\to$ `END` is present in the CFG. Then, the paths in $\mathcal{E}$ (finite paths terminating at `END`) are in a natural correspondence with the right-infinite paths in $\mathcal{R}' \subseteq \mathcal{R}$ in which only a finite number of nodes differ from `END`. In fact, postdominance could be defined by letting $\mathcal{P} = \mathcal{R}'$.

It is a straightforward consequence of Definition 6 that dominance is nondecreasing with the path set $\mathcal{P}$, in the sense that if $\mathcal{Q} \supseteq \mathcal{P}$, then $\mathcal{Q}$-dominance $\subseteq \mathcal{P}$-dominance. As a corollary (when $\mathcal{P} = \mathcal{R}'$ and $\mathcal{Q} = \mathcal{R}$), we see that loop postdominance is a subset of postdominance.

In general, inclusion between two transitive relations does not imply inclusion between their transitive reductions. Fortunately, we can show that the loop postdominance forest (denoted *lpd-forest*) is obtained by a suitable pruning of the postdominance tree (denoted *pd-tree*). This can be seen in Figure 1 for the running example. First, we introduce some notation.

- v `pd` u: v postdominates ($\mathcal{E}$-dominates) u;
- v `lpd` u: v loop postdominates ($\mathcal{R}$-dominates) u;
- `ipd`(a): $i\mathcal{E}dom(a)$, the immediate postdominator of a;
- `ilpd`(a): $i\mathcal{R}dom(a)$, the immediate loop postdominator of a.

Proposition 2 *If* `ilpd`$(a) \neq \infty$ *then* `ilpd`$(a) =$ `ipd`(a).

Proof: Let $b =$ `ilpd`$(a) \in V$. By contradiction, assume that $b \neq$ `ipd`(a). Then, there is a $c \in V$ such that b `pd` c

and c pd a. Let $\pi \in \mathcal{R}_a$ be a right-infinite path starting at a. We distinguish two cases:

- END occurs on π. Then, let π_1 be the smallest prefix of π from a to END. Since c pd a, then c occurs on π_1, hence it occurs on π and $\pi \in \mathcal{R}_c$.
- END does not occur on π. Since b lpd a, node b must occur on π. Moreover, since b pd c, there is a c-free path σ from b to END. Let π_1 be the smallest prefix of π from a to b. Then, $\pi_1\sigma$ goes from a to END and, considering that c pd a, it must contain c. Since σ is c-free, c must occur on π_1, hence it occurs on π and $\pi \in \mathcal{R}_c$.

In both cases, we see that $\mathcal{R}_a \subseteq \mathcal{R}_c$, so conclude that c lpd a. By Theorem 1, b and c are ordered by loop postdominance. Given that $b =$ ilpd(a), it must be true that c lpd b which, since loop postdominance implies postdominance, yields c pd b, in contradiction with b pd c (postdominance is acyclic). □

From Proposition 2, we see that we can build the loop postdominance tree by starting with the postdominator tree, and replacing each edge $(ipd(v) \rightarrow v)$ with $(\infty \rightarrow v)$ for all nodes v for which $ilpd(v) \neq ipd(v)$. A computationally convenient characterization of such nodes (in Figure 1, nodes b,e,f,g) is the next goal. The necessary concepts are now introduced.

Definition 10 *With reference to a CFG $G = (V, E)$ and nodes $a, w, x \in V$, we introduce the following terminology and notation:*

- *w is* prereachable *from a: there is a non trivial path from a to w not containing* ipd(a)*;*
- *x is a* crown*: x is prereachable from itself;*
- *K: the set of crowns of G;*
- *K^*: the set of nodes from which some crown is prereachable.*

Clearly, all the nodes of a path from a to w not containing ipd(a) are strictly postdominated by ipd(a). In Figure 1, e is prereachable from f because the path $f \rightarrow c \rightarrow e$ does not contain ipd$(f) = h$. Intuitively, this means e is reachable from f by a path in which all nodes lie within that subtree of the postdominator tree that is rooted at ipd$(f) = h$.

A crown is a node that lies on a cycle that does not contain its postdominator. In Figure 1, b is a crown since it lies on the cycle $a \rightarrow b \rightarrow a$, and $k =$ ipd(b) does not. Similarly, e and f are crowns since they lie on cycle $c \rightarrow e \rightarrow d \rightarrow f \rightarrow c$, and h does not. Intuitively, it is clear that if v is a crown, then $ilpd(v)$ and $ipd(v)$ are different nodes because there is an infinite path starting at v that does not contain $ipd(v)$.

Note that $ipd(g)$ is distinct from $ilpd(g)$ but g is not a crown. However, there is a path from g to crown f ($g \rightarrow d \rightarrow f$) which does not contain $ipd(g)$. In other words, $g \in K^*$. We show next that membership in K^* identifies all nodes for which $ipd(v)$ is distinct from $ilpd(v)$. Obviously, membership in K is a special case, as $K \subseteq K^*$.

Theorem 3 *For any $a \in V$,* **if** *$a \in K^*$* **then** ilpd$(a) = \infty$ **else** ilpd$(a) =$ ipd(a).

Proof: As ilpd(END) = ipd(END) = ∞, the statement is trivially true for $a =$ END. Hereafter, we assume $a \neq$ END.

Part 1: $a \in K^* \Rightarrow$ ilpd$(a) = \infty$. Consider first the case where there is a crown x prereachable from a. Let $C = x\gamma x$ be a cycle containing x but not ipd(x); such a cycle exists by Definition 10. Let σx be a path from a to x whose nodes are all strictly postdominated by $b =$ ipd(a); such a path exists by Definition 10. Clearly, $\sigma(x\gamma)^*$ -where * denotes infinite repetition- is a path in $\mathcal{R}_a - \mathcal{R}_b$, showing that $\neg(b$ lpd $a)$, Therefore, by Proposition 2, ilpd$(a) = \infty$, and the **then** clause is established.

Part 2: ilpd$(a) = \infty \Rightarrow a \in K^*$. In conjunction with Proposition 2, this establishes the **else** clause.

We let $b =$ ipd(a) and assume that $\neg(b$ lpd $a)$. By definition of loop postdominance, there is a b-free right-infinite path $\pi \in \mathcal{R}_a$ starting at a. Since π is infinite, some node must occur repeatedly. Consider the shortest prefix π_1 of π where a repetition occurs, so that π_1 has the form $\sigma y\gamma y$.

We claim that all nodes on π_1 are strictly postdominated by b. Indeed, if some z on π_1 were not postdominated by b, the prefix π_2 of π_1 terminating with z could be augmented with a b-free path τ to END. The concatenation $\pi_2\tau$ would then be a b-free path from a to END, contradicting the assumption that $b =$ ipd(a).

Consider now the cycle $C = y\gamma y$, which is part of π_1, and the nearest ancestor x of y whose parent in the pd-tree is not on C. Clearly, x is a crown prereachable from a, hence $a \in K^*$. □

To summarize, the computation of the loop postdominance forest has been reduced to that of set K^*.

In the next section, we shall first develop an algorithm to compute K from G. Then, we show how the well known connection between prereachability and conversion of a program to single static assignment form [BJP91, CFR+91, Wei92] can be exploited to compute K^* from K.

5 Optimal Computation of the Loop Postdominance Forest

We introduce a graph called the *sibling connectivity graph* associated with any CFG, which facilitates the computation of crowns. Specifically, while in the CFG a node is a crown if it lies on a cycle that does not contain the immediate postdominator of that node, in the sibling connectivity graph a node is a crown if it simply lies on a cycle.

```
1. Initialize the SCG to $(V - \{\texttt{END}\}, \emptyset)$.
2. For each node $v \in V$, create an initially empty list $L(v)$.
3. For each edge $(x \to v)$, if $v \neq \texttt{ipd}(x)$ then append $(x \to v)$ to list $L(v)$.
4. Create an initially empty stack $ST$.
5. For each node $v$ visited during a depth-first walk over the pd-tree, do:
    1. When entering $v$ for the first time, push $v$ on stack $ST$.
       For each $x \to v$ in $L(v)$, let $y$ be the node pushed immediately on top of $\texttt{ipd}(x)$ in $ST$.
       Add edge $x \to y$ to the SCG.
    2. When retreating out of $v$, pop $v$ from stack $ST$.
   {This computes the SCG.}
6. Determine the set $K_1$ of nodes with self loops in the SCG (during its construction).
7. Find the strongly connected components (scc's) of the SCG.
   Let $K_2$ be the set of nodes contained in scc's of two or more nodes.
8. Output $K = K_1 \cup K_2$ as the set of crowns.
```

Figure 2: Computing the Sibling Connectivity Graph and the Set of Crowns

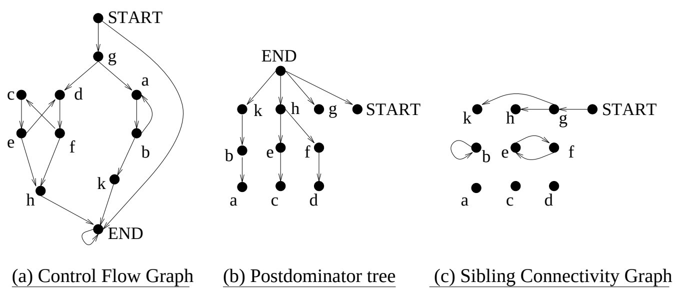


Figure 3: The Sibling Connectivity Graph

5.1 Sibling Connectivity Graph

In this subsection, we define the sibling connectivity graph and describe a linear time algorithm to compute it.

Definition 11 *With each node b of CFG $G = (V, E)$, we associate a graph $G_b = (V_b, E_b)$ where*

$$V_b = \{x \in V \ : \ \texttt{ipd}(x) = b\}$$

is the set of children of b in the pd-tree*, and*

$$E_b = \{x \to y \ : \ x, y \in V_b, \ \exists (x \to v) \in E \text{ s.t. } y \ \texttt{pd} \ v\}.$$

The collection of graphs G_b for all nodes b in the program is called the sibling connectivity graph (SCG).

An intuitive description of the SCG is the following. Let $x \to v$ be a *CFG* edge with $v \neq \texttt{ipd}(x)$. If y is the sibling of x that is an ancestor of v, the SCG has an edge $x \to y$, (a self-loop if $y = x$). For example, in Figure 1, edge $(g \to a) \in E$, and k is the sibling of g that is an ancestor of a; therefore, edge $(g \to k)$ occurs in the SCG. Figure 3 shows the SCG for the running example; component G_h consists of the set of nodes $\{e, f\}$ and the edges between these nodes.

Proposition 3 *The SCG corresponding to a CFG $G = (V, E)$ can be constructed in time $O(|V| + |E|)$.*

Proof: The procedure is shown in Figure 2, steps 1-5. A depth-first walk is performed over the postdominator tree. First, we construct a list L of edges at each node v : if $x \to v$ is a *CFG* edge and v is not $\texttt{ipd}(x)$, then this edge is entered into list $L(v)$. During the depth-first walk, we maintain a stack of nodes such that when the walk is at a node n, the stack consists of the ancestors of n ordered by ancestorship. This is accomplished by pushing a node n on the stack when the walk first reaches n, and popping it from the stack when retreating out of node n. When the walk reaches a node v, we examine the list of edges $L(v)$: if $x \to v$ is an edge in this list, we first locate the parent of x in the postdominator tree (say p), and then find the node pushed immediately on top of p in the stack (say y). The SCG has an edge $x \to y$. The stack can be implemented as a doubly-linked list to support this 'find' operation (as well as pushing and popping) in constant time. The correctness of this procedure follows trivially from properties of depth-first search. It is also easy to show that the entire procedure takes time proportional to the number of vertices and edges in the CFG. □

5.2 Crowns

In this subsection, we characterize the crowns with respect to the SCG and develop a fast algorithm to compute the set K of all crowns.

Proposition 4 *With the notation of Definition 11, $y \in V_b$ is* prereachable *from $x \in V_b$ in G if and only if y is* reachable *from x in G_b.*

Proof: Assume first that y is prereachable from x is G, say via path π. Decompose π as $u_0\pi_1u_1\pi_2...u_{r-1}\pi_ru_r$, with $u_0 = x$ and $u_r = y$, where $u_0...u_r$ are those nodes on π that belong to V_b, while the nodes on π_i are descendant of u_i in the *pd-tree*. Such decomposition (possibly, with some of the π_i's empty) is always possible because:

1. by definition of prereachability, all nodes on π are strictly postdominated by b, and
2. if edge $u \to v$ is on π, then whenever u and v are descendant of different children of b in the *pd-tree*, then u must be a child of b (by the characterization of CFG edges with respect to the *pd-tree*).

Now, let v_i be the node immediately following u_i on π. Then, $(u_{i-1} \to v_i) \in E$ implies that $(u_{i-1} \to u_i) \in E_b$. Hence, $(x = u_0)u_1...u_{r-1}(u_r = y)$ is a path from x to y in G_b.

For the converse, assume now that y is reachable from x in G_b, say via a path $\sigma = u_0u_1...u_r$, with $u_0 = x$ and $u_r = y$. Then, for $i > 0$, by Definition 11 of E_b, there is a descendant v_i of u_{i+1} in the *pd-tree* such that $(u_i \to v_i) \in E$. Clearly, u_{i+1} is reachable (in G) from v_i via a path π_i entirely postdominated by b (since u_{i+1} `pd` v_i). Hence, $u_0\pi_1...\pi_r$ is a path from $u_0 = x$ to $u_r = y$. □

As a corollary of the preceding proposition, we have the following characterization of the crowns.

Proposition 5 *A node x is a crown if and only if it lies on some cycle (possibly, a self loop) of the SCG.*

Proof: Assume first that x is a crown of cycle C, in G, and let $(x \to v) \in E$ be the edge of C emanating from x.

If x `pd` v, **then**, by Definition 11 of $E_{\texttt{ipd}(x)}$, $G_{\texttt{ipd}(x)}$ contains the self loop $x \to x$ as stated.

Else, let y ($\neq x$) be the sibling of x among the ancestors of v. It is easy to see that y is also on C, so that x and y are reachable from each other, in G. Moreover, since all nodes of C are strictly postdominated by $\texttt{ipd}(x)$, x and y are prereachable from each other, in G. By Proposition 4, x and y are reachable from each other in $G_{\texttt{ipd}(x)}$, hence x lies on a cycle of $G_{\texttt{ipd}(x)}$.

For the converse, assume that x lies on a cycle C' of $G_{\texttt{ipd}(x)}$.

If C' is a self loop $x \to x$, **then**, by Definition 11, there is an edge $(x \to v) \in E$ where x `pd` v. As there is always a path, say $v\pi x$, from a node v to a postdominator of it x, we can construct the cycle $C = xv\pi x$ in G having x as a crown.

Else C' contains at least a node $y \neq x$. Clearly, x and y are reachable from each other in $G_{\texttt{ipd}(x)}$ so that, by Proposition 4, they a prereachable from each other in G, say, via paths $x\sigma y$ and $y\tau x$. Then, $C = x\sigma y\tau x$ is a cycle in G whose nodes are all postdominated by $\texttt{ipd}(x)$ $(= \texttt{ipd}(x))$, whence x (as well as y) is a crown of C. □

Proposition 6 *The set K of the crowns of a CFG $G = (V, E)$ can be computed in time $O(|V| + |E|)$.*

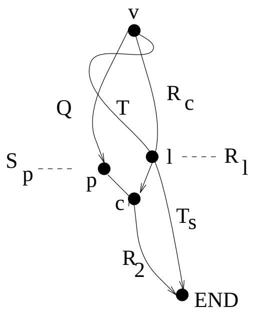


Figure 4: Paths in Lemma 1

Proof: The complete procedure is shown in Figure 2. Having constructed the SCG (Proposition 3), its strongly connected component (scc's) are computed in $O(|V| + |E|)$ time by the algorithm of Tarjan [Tar72], (see also [CLR92]). Based on Proposition 5, the crowns of G can be determined by identifying those nodes of the SCG that lie either on a self loop or on a cycle (which is equivalent to membership in a scc of size at least two). Both conditions are easily checked. □

5.3 Prereachability to Crowns

Finally, we must compute the set K^* of those nodes from which some crown is prereachable.

Our approach consists in reducing this problem to that of finding the single static assignment (SSA) form of a program whose CFG is the *reverse* of $G = (V, E)$, that is, $G_R = (V, E_R)$, where $E_R = \{v \to u : (u \to v) \in E\}$. We show that if K represents the set of assignments to some dummy variable X, then K^* is precisely the set of the so-called *join nodes* [CFR+91] where ϕ-functions must be introduced to convert G_R to SSA form (with respect to X).

We begin by recalling the definition of join nodes [CFR+91], in a form that is convenient for the present developments.

Definition 12 *Let $S \subseteq V$. The set $J_R(S)$ of* join nodes *for S in G_R is the set of all nodes z such that there are in G two non trivial paths $z \xrightarrow{+} x_1$ and $z \xrightarrow{+} x_2$, with $x_1, x_2 \in S$, intersecting only at z.*

The following lemma is the key step to make the connection between prereachability in G and join nodes in G_R.

Lemma 1 *With reference to CFG G, if c is prereachable from v, then there exist paths $P_1 = v \xrightarrow{+} c$ and $P_2 = v \xrightarrow{+}$ `END` intersecting only at v.*

Proof: The proof is an induction on the length of the shortest non trivial path $P = v \xrightarrow{+} c$ which establishes the prereachability of c from v.

Suppose the length of P is 1, i.e., $P = vc$. Since c does not postdominate v, there is a path $P_2 = v \xrightarrow{+} \texttt{END}$ not containing c and hence disjoint from $P_1 = P$, except for node v.

Assume the lemma is true for paths of length less than n. Let $P = v \xrightarrow{+} p \rightarrow c$ be a path of length n. By the inductive assumption, there is a path $R_2 = v \xrightarrow{+} \texttt{END}$ that is disjoint, except for node v, from a path $S_p = v \xrightarrow{+} p$. WLOG, assume that R_2 is acyclic. Appending the edge $p \rightarrow c$ to path S_p gives a path Q from v to c. If c does not occur on path R_2, or if $c = v$, the lemma is proved by setting $P_1 = Q$, and $P_2 = R_2$.

Otherwise, c occurs on R_2 and is distinct from v. Let R_c be the prefix $v \xrightarrow{+} c$ of R_2. Note that Q and R_c are two paths from v to c that are disjoint except for v and c. Consider an acyclic path $T = v \xrightarrow{+} \texttt{END}$ that does not contain c; such a path must exist because c does not strictly postdominate v. Let l be the last node on T that occurs on either Q or R_c — that is, the suffix $T_s = l \xrightarrow{+} \texttt{END}$ of path T is disjoint (other than node l) from paths Q and R_c (l must exist because all three paths contain v).

If $l = v$, then let $P_1 = Q$ and $P_2 = T$.

Otherwise, l is distinct from v, and it is contained in exactly one of paths R_c and Q. Suppose l is contained in R_c. Path R_c can be written as $v \xrightarrow{+} l \xrightarrow{+} c$ where the prefix $v \xrightarrow{+} l$ is called R_l. Concatenate R_l and T_s to get a path $P_2 = v \xrightarrow{+} \texttt{END}$ which is disjoint from $P_1 = Q$. The case when l is contained in Q is identical. □

We can now make the connection to SSA computation.

Theorem 4 *Let $S \in V$ with $\texttt{END} = \texttt{START}_R \in S$. Then, $v \in J_R(S)$ if and only if there is a $c \in S$ that is prereachable from v in G.*

Proof: ($\Rightarrow$) By Definition 12 of $J_R(S)$, there are two distinct nodes c_1 and c_2 in S and two non trivial paths $v \xrightarrow{+} c_1$ and $v \xrightarrow{+} c_2$ in G intersecting only at v. At least one of these two paths does not contain $\texttt{ipd}(v)$, thus establishing the prereachability of its endpoint (c_1 or c_2) from v in G.

($\Rightarrow$) From Lemma 1, there are two paths $v \xrightarrow{+} \texttt{END}$ and $v \xrightarrow{+} c$ in G that are disjoint except for v. Since c and END are both in S, by Definition 12, $v \in J(S)$. □

In our running example, the crowns are $\{b, e, f\}$; if these nodes and END are treated as assignments to some variable in the *reverse* CFG G_R, we need ϕ-functions at nodes $\{b, e, f, g\}$. The set of nodes $\{b, e, f, g\}$ is precisely the set of nodes that in G are *not* loop postdominated by their immediate postdominator.

It is worth observing that, while in general S is not necessarily a subset of $J_R(S)$, it is the case that $K \subseteq J_R(K)$ when K is the set of crowns. This is because (i) END is a crown (due to the selfloop $\texttt{END} \rightarrow \texttt{END}$) and each crown is prereachable from itself (by Definition 10).

In conclusion, $K^* = J_R(K)$, which can be computed in $O(|E|)$ time by any of several SSA algorithms in the literature [SG95, PB95], such as the one described in our earlier work on APT [PB95].

5.4 Summary

The following theorem summarizes the result of our approach to loop control dependence computations:

Theorem 5 *Given a CFG $G = (V, E)$ containing the edge $\texttt{END} \rightarrow \texttt{END}$, the corresponding Augmented Loop Postdominator Tree can be constructed in linear time and stored in linear space. It can answer loop control dependence queries of the* `cd`, `conds`, *and* `cdequiv` *types in time proportional to the size of their answers.*

Proof: Figure 5 summarizes the procedure developed in this paper. The linear bound follows: for Step 1, by [Har85]; for Step 2, by Proposition 6; for Step 3, by the known results on SSA (e.g., [PB95]); for Steps 4 and 5, by straightforward procedures; for Step 6, by the preprocessing algorithms for the $\mathcal{APT}$ data structure presented in [PB95]. □

6 Conclusions

We have presented a framework, based on a generalized notion of dominance, that permits a uniform treatment of classical and loop control dependence. We have applied this framework to compute the weak (or loop) control dependence relation optimally.

It would be interesting to include Ballance and McCabe's *hierarchical control dependence* [BM92] in our framework. Unfortunately, this relation has been defined by specifying a procedure for computing it. The formulation of this relation in graph-theoretic terms is a prerequisite for fitting it into our framework.

Acknowledgements: We are obligated to Tom Reps for pointing us to the work of Podgurski and Clarke. Andy Podgurski was kind enough to send us a copy of his dissertation at short notice. We also thank Ron Cytron, Jeanne Ferrante, Bjaarne Steensgard and Mike Wolfe for useful discussions on control dependence. Finally, we would like to mention that we found the PLDI '96 referee reports very useful in revising the extended abstract.

References

[BJP91] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

1. Build the postdominator tree *pd-tree* of the given CFG $G = (V, E)$, including the distinguished node ∞ as described in Subsection 3.1.
2. Compute the set of crowns K as shown in Figure 2.
3. In the reverse CFG G_R, mark all crowns (K) as assignments to some dummy variable X, and perform an SSA computation to obtain the set K^* of nodes where the SSA form will require ϕ-functions for variable X.
4. For each $a \in K^*$, replace pd-tree edge ($\texttt{ipd}(a) \rightarrow a$) with lpd-tree edge ($\infty \rightarrow a$).
5. For each edge $(u \rightarrow v) \in E$ such that v does not loop postdominate u, append to (an initially empty) route array A a chariot route with end points v and $\texttt{ilpd}(u)$.
6. Construct the $\mathcal{APT}$ for the Roman Chariot Problem in which the tree is the loop postdominator tree and the route array is A.

Figure 5: Computing the Loop Control Dependence Relation

[BM92] Robert Ballance and Arthur McCabe. Program dependence graphs for the rest of us. Technical Report 92-10, University of New Mexico, October 1992.

[CFR+91] 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.

[CFS90] Ron Cytron, Jeanne Ferrante, and Vivek 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.

[CLR92] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1992.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependency graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, June 1987.

[Har85] D. Harel. A linear time algorithm for finding dominators in flowgraphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 185–194, Providence, Rhode Island, May 6–8, 1985.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *Siam Journal of Computing*, 13(4):338–355, 1984.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[PB95] Keshav Pingali and Gianfranco Bilardi. APT: A data structure for optimal control dependence computation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 1995.

[PC90] Andy Podgurski and Lori Clarke. A formal model of program dependences and its implications for software testing, debugging and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, Septmeber 1990.

[Pod89] Andrew Podgurski. *The significance of program dependences for software testing, debugging and maintenance.* PhD thesis, University of Massachusetts, Amherst, 1989.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, San Francisco, California, January 1995.

[Tar72] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[Wei92] Michael Weiss. The transitive closure of control dependence: The iterated join. *ACM Letters on Programming Languages and Systems*, 1(2):178–190, June 1992.