$\mathcal{APT}$: A Data Structure for Optimal Control Dependence Computation

Keshav Pingali [1]
Department of Computer Science
Cornell University, Ithaca, NY 14853.

Gianfranco Bilardi [2]
Dipartimento di Elettronica ed Informatica,
Università di Padova, Padova, Italy.

Abstract

The *control dependence* relation is used extensively in restructuring compilers. This relation is usually represented using the *control dependence graph*; unfortunately, the size of this data structure can be quadratic in the size of the program, even for some structured programs. In this paper, we introduce a data structure called the *augmented postdominator tree* ($\mathcal{APT}$) which is constructed in space and time proportional to the size of the program, and which can answer control dependence queries in time proportional to the size of the output. Therefore, $\mathcal{APT}$ is an *optimal representation of control dependence*. We also show that using $\mathcal{APT}$, we can compute SSA graphs, as well as sparse dataflow evaluator graphs, in time proportional to the size of the program. Finally, we put $\mathcal{APT}$ in perspective by showing that it can be viewed as a factored representation of the control dependence graph in which *filtered search* is used to answer queries.

1 Introduction

Control dependence is a key concept in program optimization and parallelization. Intuitively, a node n is control dependent on a node c if c determines whether n is executed. For example, in a conditional statement, statements on the true and false sides of the conditional are control dependent on the predicate. Since statements from opposite sides of the conditional statement are control dependent on the predicate in a complementary sense, it is more precise to say that a node is control dependent on an *edge*; for example, we can say that statements on the true side of a conditional statement are control dependent on the true edge from the predicate. In the presence of nested control structures, multiway branches and unstructured flow of control, intuition is an unreliable guide, and it is better to use a formal, graph-theoretic definition of control dependence. This definition requires the following concepts.

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

To simplify the discussion, we will follow standard practice and assume that there is an edge from `START` directly to `END` in the control flow graph [FOW87].

Definition 2 *A node* w *is said to* **postdominate** *a node* v *if every path from* v *to* `END` *contains* w*.*

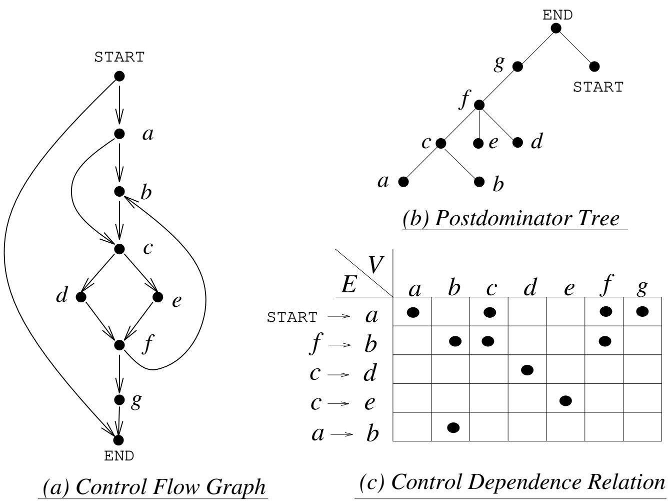
Note that a node v is always postdominated by `END` and by itself. It can be shown that postdominance is a transitive relation, and that its transitive reduction is a tree-structured relation called the *postdominator tree*. The parent of a node in this tree is called the *immediate* postdominator of that node. The postdominator tree of a program can be constructed in $O(|E|\alpha(|E|))$ time using an algorithm due to Tarjan and Lengauer [LT79], or in $O(|E|)$ time using a rather more complicated algorithm due to Harel [Har85]. Control dependence can be defined formally as follows [FOW87]:

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

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

[1] Keshav Pingali (pingali@cs.cornell.edu) was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

[2] 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.



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

Figure 1: A Program and its Control Dependence Relation

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, it is possible to reach END from u without passing through w. Thus, u is a 'decision-point' that determines whether w will be executed. Let C denote the set of all pairs (e, w) such that node w is control dependent on edge e. We refer to the set C ($\subseteq E \times V$) as the *control dependence relation* of the control flow graph G.

Control dependence is used in many phases of modern compilers, such as dataflow analysis, loop transformations and code scheduling. An abstract view of these applications is that they require the computation of the following sets derived from C [CFS90]:

- $\texttt{cd}(e) = \{w \in V | (e, w) \in C\}$,
- $\texttt{conds}(w) = \{e \in E | (e, w) \in C\}$, and
- $\texttt{cdequiv}(w) = \{v \in V | \texttt{conds}(v) = \texttt{conds}(w)\}$

Set $\texttt{cd}(e)$ is the set of nodes that are control dependent on edge e, while $\texttt{conds}(w)$ is the set of control dependences of node w. These sets are used in scheduling instructions across basic block boundaries for speculative or predicated execution [Fis81, BR91]. They are also useful in merging program versions [HPR87], and in automatic parallelization [ABC$^+$88]. Set $\texttt{cdequiv}(w)$ contains the nodes that have the same control dependences as node w. 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 [GS87]. The relation cdequiv 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 [JPP94, Joh94], and to perform dataflow analysis in parallel [JPP94, GPS90]. Figure 1 shows a small program and its control dependence relation. For any edge e, $\texttt{cd}(e)$ is the set of marked nodes in the row corresponding to e. For any node w, $\texttt{conds}(w)$ is the set of marked edges in the column corresponding to w. Finally, we see that $\texttt{cdequiv}(c) = \texttt{cdequiv}(f) = \{c, f\}$, and $\texttt{cdequiv}(a) = \texttt{cdequiv}(g) = \{a, g\}$; all the other nodes are in cdequiv sets by themselves.

In this paper, we design a data structure to represent the control dependence relation. Such a data structure must be evaluated along three dimensions:

- *preprocessing time T*: the time required to build the data structure,
- *space S*: the overall size of the data structure, and
- *query time Q*: the time required to produce the answer to cd, conds and cdequiv queries.

The size of the control dependence relation gives an upper bound on the space requirements of such a data structure. It is easy to show that the size of the relation is $\Omega(|V||E|)$, even if we restrict our attention to structured programs. For example, for programs consisting of n nested repeat-until loops, it can be verified that $|E| = \Omega(n)$ and $|C| = \Omega(n^2)$; therefore, the size of the control dependence relation can grow quadratically with program size. It would be incorrect to conclude that quadratic space is a *lower* bound on the size of any representation of the control dependence relation. Note that the size of the postdominator relation grows quadratically with program size (consider a chain of n nodes), but this relation can be represented using the postdominator tree, which can be built in $O(|E|)$ space [LT79, Har85], and which provides proportional time access to the postdominators of a node. The explanation of the paradox is that postdominance is a transitive relation, and the postdominator tree, which is the transitive reduction of this relation, is a 'factored', compact representation of postdominance. There is no point in building a representation of the full relation because the factored relation is more compact, and it answers queries optimally.

Is there a factored representation of the control dependence relation which can be built in $O(|E|)$ space and $O(|E|)$ preprocessing time, and which will answer cd, conds and cdequiv queries in time proportional to the size of the output?

The standard representation of the control dependence relation is the *control dependence graph* (CDG) [FOW87], which is best viewed as a bipartite graph in which the two sets of nodes in the bipartite graph are V and E, and in which there is an undirected edge between node v and edge e if v is control dependent on e. Since this is a straight-forward representation of the full relation, the size of the CDG is $\Omega(|E||V|)$. There have been many efforts to construct more compact representations of the control dependence relation [FOW87, CFS90, Bal93, JP93, SGL94], and the lack of success led Cytron, Ferrante and Sarkar to conjecture that any data structure that provided proportional time access to control dependence sets must use space that grows quadratically with program size [CFS90].

In this paper, we describe a data structure called the *aug-*

mented postdominator tree ($\mathcal{APT}$) which requires $O(|E|)$ space, is built in $O(|E|)$ time[3], and which is designed to provide proportional time access to `conds` sets. This is clearly optimal to within a constant factor. In fact, our approach incorporates a design parameter $\alpha(>0)$, under the control of the compiler writer, representing a trade-off between time and space. A smaller value of α results in faster query time at the expense of more memory for a larger data structure. Interestingly, the control dependence graph can be viewed as one extreme of this range of data structures, obtained when α is less than $1/|E|$. Using existing algorithms for `cd` and `cdequiv`, we show that the $\mathcal{APT}$ data structure provides proportional time access to `cd` and `cdequiv` sets as well, completing the description of an optimal data structure for control dependence.

The rest of the paper is organized as follows. In Section 2, we reformulate the `conds` problem as a naturally stated graph problem called the Roman Chariots problem. In Section 3, we describe the $\mathcal{APT}$ data structure and the algorithm for the `conds` problem. The data structure is a minor augmentation of the postdominator tree; the query procedure for `conds` sets performs a walk over parts of the postdominator tree. We also show how $\mathcal{APT}$ can be used to answer `cd` and `cdequiv` queries. Some experimental results on a model problem and on the SPEC benchmarks are reported Section 4. In Section 5, we discuss an application of these techniques: we show how to perform ϕ-function placement, the key step in SSA computation [CFR$^+$91], in $O(|E|)$ time. Finally, in Section 6, we contrast our approach with dynamic techniques like memoization [Mic68]; we also show that our approach can be viewed as an example of Chazelle's *filtered search* [Cha86].

2 The Roman Chariots Problem

We reformulate the control dependence problem as a naturally stated graph problem called the Roman Chariots problem, using the fact that nodes that are control dependent on an edge e in the control flow graph form a simple path in the postdominator tree [FOW87]. First, we introduce some convenient notation.

Definition 4 *Let $T =< V, F >$ be a tree. For $v, w \in V$, the notation $[v, w]$ represents the set of vertices on the simple path joining v and w in P. Similarly, the notation $[v, w)$ represents the set of vertices on the simple path joining v and w in P, not including w. (In particular, $[v, v)$ is empty.)*

For example, in the postdominator tree of Figure 1(b), $[d, a]$ denotes the set $\{d, f, c, a\}$, while $[d, a)$ represents the set $\{d, f, c\}$. This notation is similar to the standard mathematical notation for open and closed intervals of the line.

[3] We assume that the postdominator relation is computed using Harel's algorithm; if the Lengauer and Tarjan algorithm is used, preprocessing time becomes $O(|E|\alpha(|E|))$.

The following key theorem is due to Ferrante, Ottenstein and Warren [FOW87].

Theorem 1 *If $(u \to v)$ is an edge of the control flow graph, then*

1. *$parent(u)$ is an ancestor of v in the postdominator tree, and*
2. $\texttt{cd}(u \to v) = [v, parent(u))$.

Proof: Note that since no control-flow edge emanates from END, the expression $parent(u)$ is defined whenever $(u \to v) \in E$.

1. If $parent(u)$ does not postdominate v, we can find a path $v \to ... \to$ END which does not contain $parent(u)$. Prefixing this path with the edge $u \to v$, we obtain a path from u to END which does not contain $parent(u)$, contradicting the fact that $parent(u)$ postdominates u.
2. We show that $\texttt{cd}(u \to v) \subseteq [v, parent(u))$. Let w be an element of $\texttt{cd}(u \to v)$. From the definition of control dependence, w must postdominate v, so w is on the path $[v, \texttt{END}]$ in the postdominator tree. From part (1), $parent(u)$ is also on the path $[v, \texttt{END}]$. However, w cannot be on the path $[parent(u), \texttt{END}]$ since in that case, it would be distinct from u and postdominate u. Therefore, w must be on the path $[v, parent(u))$.

   Conversely, assume that w is contained in the path $[v, parent(u))$. From part (1), it follows that w postdominates v; it also follows that w does not postdominate $parent(u)$. Therefore, if $w \neq u$, then w cannot postdominate u either. Therefore, w is control dependent on edge $u \to v$.

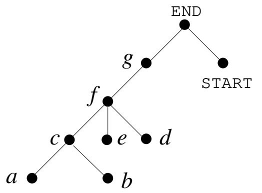
□

Figure 2 shows the non-empty `cd` sets for the program of Figure 1(i). If $[v, w)$ is a `cd` set, we will refer to v and w as the *bottom* and *top* nodes of this set respectively, where the orientations of bottom and top are with respect to the tree[4]. The postdominator tree and the array of `cd` sets, together, can be viewed as a compact representation of the control dependence relation since we can recover the full control dependence relation by expanding each entry of the form $[v, w)$ to the corresponding set of nodes by walking up the postdominator tree from v to w. The advantage of using the postdominator tree and `cd` sets, instead of the CDG, is that they can be represented in $O(|E|)$ space, and as we will see, can be built in $O(|E|)$ time. What is not obvious is how they can be used to answer control dependence queries in proportional time — that is the subject of the rest of the paper.

[4] As an aside, we remark that the bottom-closed, top-open representation for the sets has been chosen here since it is the most immediate to obtain in our application. In general, a closed set$[b, t]$, in which t is an ancestor of b, is readily converted into the equivalent half-open one $[b, parent(t))$, in constant time. The conversion of set $[b, t)$ into a closed one is less straightforward, and takes time proportional to the number of children of t, assuming that ancestorship can be decided in constant time. However, if the conversion has to be performed for a batch of half-open sets A, it can be accomplished in time $O(|V| + |A|)$ by a depth-first traversal of the tree. We do not need this conversion.

E \ V	a	b	c	d	e	f	g
START → a	●		●			●	●
f → b		●	●			●	
c → d				●			
c → e					●		
a → b		●					

(a) Control Dependence Relation



E	cd *sets*
START $\rightarrow a$	[a, END)
$f \rightarrow b$	[b,g)
$c \rightarrow d$	[d,f)
$c \rightarrow e$	[e,f)
$a \rightarrow b$	[b,c)

(b) Postdominator Tree and cd *sets*

Figure 2: Compact Representation of Control Dependence

```
Procedure ConstructRomanChariots(G:CFG):Tree, RouteArray;
{
    % G is the control flow graph,
    T := build-postdominator-tree(G);
    A := [ ]; %Initialize to empty array
    i := 0;
    for each node u in T in top-down order do
    for each child c of u do
      for each edge (c → v) in G do
        if v is not u
          then %append a cd set to end of A
          i := i + 1;
          A[i] := [v, u);
          Note the correspondence between
          edge c → v and index i;
        endif
      od
    od
    return T, A;
}
```

Figure 3: Constructing a Roman Chariots Problem

For the purpose of exposition, it is convenient to assume that the array of cd sets, which is indexed by CFG edges in Figure 2, is indexed instead by the integers $1..m$, where m is the number of CFG edges for which the corresponding cd sets are non-empty. We will assume that the conversion from an integer (between 1 and m) to the corresponding CFG edge and vice versa can be done in constant time. We can now reduce the control dependence problem to a naturally stated graph 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 of the tree is Rome* [5]. *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 as $[v, w)$, where v is a descendant of w in T, design a data structure to answer the following queries optimally.*

1. cd(n): *Enumerate the cities on route n.*
2. conds(w): *Enumerate the set of routes that serve city w.*
3. cdequiv(w): *Enumerate the set of cities that are served by all and only the routes that serve city w.*

The control dependence problem is reduced to the Roman Chariots problem as follows. Procedure **ConstructRomanChariots** in Figure 3 takes a control flow graph as input, and returns the corresponding Roman Chariots problem. Assuming the postdominator tree can be built in time $O(|E|)$, Procedure **ConstructRomanChariots** takes time $O(|E|)$, and space $O(|E|)$. Control dependence queries are handled as follows.

- cd$(u \rightarrow v)$: If v is $parent(u)$, return the empty set. Otherwise, let i be the index into array A for edge $u \rightarrow v$. Execute the Roman Chariots query cd(i).
- conds(w): Execute the Roman Chariots query conds(w), and translate each integer (between 1 amd m) returned by this query to the corresponding CFG edge.
- cdequiv(w): Execute the corresponding Roman Chariots query cdequiv(w).

The correctness of this reduction follows immediately from Theorem 1 and Procedure **ConstructRomanChariots**. In the construction of Figure 3, the cd sets in A are sorted by decreasing top nodes; that is, if t_1 is a proper ancestor of t_2 in the postdominator tree, then any cd set whose top node is t_1 is inserted in the array before any cd set whose top node is t_2. We will exploit this order when we build the $\mathcal{APT}$ structure in Section 3. Note that for a general Roman Chariots problem (not arising from a control dependence problem), this sorting can be done by a variation of Procedure **ConstructRomanChariots**, in time $O(|A| + |V|)$. This is within the budget for preprocessing time given below. Therefore, we will assume without loss of generality that A has been sorted in this way.

The rest of the paper establishes the following result.

[5] A thorough literature search failed to turn up any historical evidence to support this statement, but it is a matter of record that all roads led to Rome [CicBC], just as in a tree rooted at Rome!

Theorem 2 *There is a data structure $\mathcal{APT}$ for a Roman Chariots problem $(T =< V, F, ROME >, A)$ which can be constructed in time $T = O(|A| + (1 + 1/\alpha)|V|)$, and stored in space $S = O(|A| + (1 + 1/\alpha)|V|)$, where $\alpha > 0$ is a design parameter. By traversing $\mathcal{APT}$, the following queries can be answered.*

- $\texttt{cd}(e)$*: Query answered in time proportional to output size. The time is independent of α.*
- $\texttt{conds}(w)$*: Query answered in time $O((1 + \alpha)s)$ where s is output size.*
- $\texttt{cdequiv}(w)$*: Query answered in time proportional to output size. The time is independent of α.*

For the special case of a tree which is the postdominator tree of a control flow graph, (i) $|A| \leq |E|$, and (ii) $|V| \leq |E|+1$, since the control flow graph is connected. Therefore, we have the following result.

Corollary 1 *Given a CFG $G = (V, E)$, structure $\mathcal{APT}$ can be built in $O(|E|)$ preprocessing time and space, and provides proportional time access to* `cd`*,* `conds` *and* `cdequiv` *sets.*

3 $\mathcal{APT}$: Solving the Roman Chariots Problem Optimally

The $\mathcal{APT}$ data structure to solve a Roman Chariots problem (T, A) is an augmentation of the tree T, and is described incrementally in this section.

3.1 `cd` queries

`cd` queries are easy: if the query is $\texttt{cd}(i)$, where i is between 1 and m (the size of A), let $[v, w)$ be the i^{th} route in A. Walk up the tree T from node v to node w, and output all nodes encountered in this walk, other than node w. This takes time proportional to the size of the output. This algorithm is similar to one due to Ferrante *et al* [FOW87].

3.2 `conds` queries

One way to answer `conds` queries is to examine all routes in array A, and report every route whose bottom node is a descendant of the query node, and whose top node is a proper ancestor of the query node. This requires examining all routes in A, which is too slow.

A better approach is to limit the search to routes whose bottom nodes are descendants of the query node, since these are the only routes that can contain the query node. To facilitate this, we will assume that at every node v, we have recorded all routes whose bottom node is v; then, the query procedure must visit the subtree of the postdominator tree rooted at the query node, and examine routes recorded at these nodes. This is shown in Figure 4(a). The space taken by the data structure is $S = O(|V| + |A|)$, which is optimal. However, in the worst case, the query procedure must examine all nodes and all routes (consider the query $\texttt{conds}(\texttt{END})$), so query time is $Q = O(|A| + |V|)$, which is too slow. To speed up query time, we extend this idea as follows. Rather than store a route only at its bottom node, we can store the route at *every* node contained in the route, as in Figure 4(b). Given a query at node q, the query procedure simply outputs all routes stored at that node; if $|A_q|$ is the size of this output, this takes time $Q = O(|A_q|)$, which is optimal. Unfortunately, this strategy produces the control dependence graph in disguise, and therefore blows up space requirements. For example, for the Roman Chariots problem arising from a nested repeat-until loop, the reader can verify that $\Omega(|A|)$ routes each contain $\Omega(|V|)$ nodes and hence are represented in as many lists, requiring space $S = \Omega(|V||A|)$ overall, which is far from optimal.

These considerations suggest that there is a space-time tradeoff in computing `conds` sets — 'caching' information at nodes on a route reduces query time, but increases storage requirements. Let us call the two extremes of caching *no caching* (store a route only at its bottom node), and *full caching* (store a route at every node contained in the route). To explore the trade-off, we consider caching information at some but not necessarily all nodes on a route. Informally, we can say that the nodes in the tree are partitioned into *zones*, and all the nodes in a zone share cached information in the sense that even if a route contains two or more nodes from a zone, it is cached at exactly one node in that zone. In Figure 4(c), there are six zones induced by the following sets of nodes: $\{a, b, c\}$, $\{d\}$, $\{e\}$, $\{f, g\}$, $\{\texttt{START}\}$ and $\{\texttt{END}\}$. Note that even though Chariot Route 1 contains both nodes a and c, it is stored only at node a; similarly, it contains both f and g but is stored only at node f.

To quantify the space-time trade-off, we must define zones formally. We require that a zone be formed by a subset of nodes in the tree, such that these nodes and the edges in T between them form a tree. For simplicity, we will require that any node v in a zone can be classified either as an *interior* node, which means that v and its children are all in the same zone, or as a *boundary* node, which means that v and its children are all in different zones; by convention, the leaves of the tree are boundary nodes. Intuitively, this rules out the possibility that some but not all the children of a node are in the same zone as itself. In Figure 4, boundary nodes are shown as solid dots, while interior nodes are shown as hollow dots; for example, in Figure 4(c), nodes c and g are interior nodes, while all other nodes are boundary nodes. The details of our zone partitioning algorithm will be given in Section 3.2.1.

Assuming that zones are given to us, we can define how routes are cached in the tree. With each node $v \in V$, we associate a list of routes $L[v]$, defined formally as follows.

Definition 5 *If v is an interior node, $L[v]$ is the list of all routes whose* bottom node *is v; if v is a boundary node, $L[v]$ is the list of all routes* containing *v.*

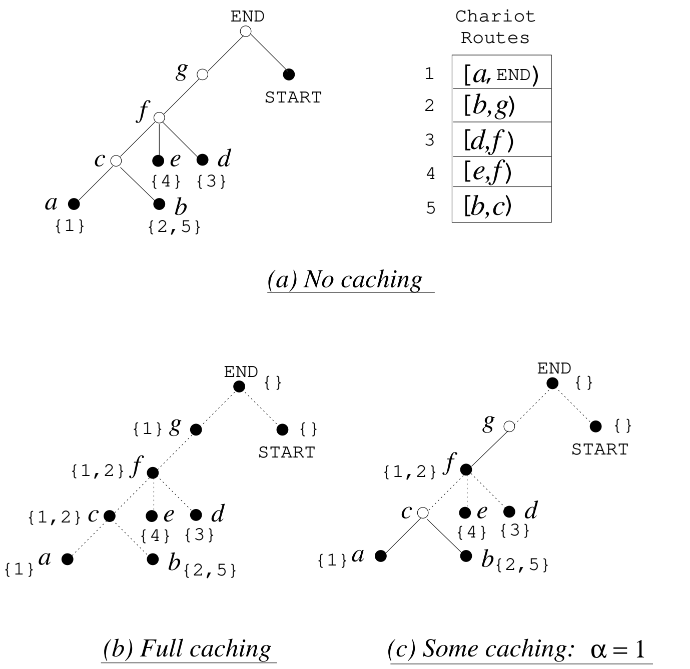


Figure 4: Zone Structure

The motivation for this definition becomes obvious if we think about the query procedure. When a node q is queried, the query procedure visits all descendants of q that are in the same zone as q (call this set Z_q), and for each visited node v, reports all routes in $L[v]$ whose top endpoint t is a proper ancestor of q. For example, in Figure 4(c), a query at node g results in visits to nodes g and f. Using the definition of $L[v]$, we can see that $A_q \subseteq \cup_{v \in Z_q} L[v]$. A route containing a given query node q must originate at a descendant b of q; so if $b \in Z_q$, then the route is in $L[b]$, and otherwise, the route enters Z_q through some boundary node w, and is in $L[w]$. To avoid examining routes unnecessarily, we will assume that each list $L[v]$ is sorted by top end point, from higher (closer to the root) to lower. Examination of routes in a list $L[v]$ can terminate as soon as a route $[b, t)$ not containing q is encountered; further routes on the list terminate at a descendant of t and do not contain the query node q.

It follows immediately that the query time is proportional to the sum of the number of visited nodes and the number of reported routes. If we define the *subzone* Z_q associated with a node q to be the set of descendants of q that are in the same zone as q. The query time for node q can be written as follows:

$$Q_q = O(|A_q| + |Z_q|). \tag{1}$$

A simple implementation of this query procedure is given in Figure 5. Boundary nodes are distinguished from interior nodes by a boolean named $Bndry$? which is set to true for boundary nodes and to false for interior nodes; the algorithm for determining which nodes are boundary nodes will

```
Procedure Query(QueryNode);
{
1:    % APT data structure is global variable;
2:    % Query outputs list of routes numbers
3:    Visit(QueryNode, QueryNode);
}
Procedure Visit(QueryNode, VisitNode);
{
1:    for each route i in L[VisitNode]
2:      in list order do
3:        let A(i) be [b, t);
4:        if t is a proper ancestor of QueryNode
5:        then output i;
6:        else break ; % exit from the loop
7:      od ;
8:    if VisitNode is not a boundary node
9:      then
10:       for each child C of VisitNode
11:         do
12:           Visit(QueryNode,C)
13:         od ;
14:   endif ;
15:
}
```

Figure 5: Query Procedure for `conds`

be described in Section 3.2.1. In line 4 of Procedure **Visit**, testing whether t is a proper ancestor of $QueryNode$ can be done in constant time as follows: since t and $QueryNode$ are ordered by the ancestor relation, we can give each node a dfs (depth-first search) number, and establish ancestorship by comparing dfs numbers. Since *dfs* numbers are already assigned by postdominator tree construction algorithms [LT79, Har85], this is convenient. Alternatively, we can use level numbers in the tree.

3.2.1 Algorithm for Determining Zones

Let us first require that the following inequality holds for all nodes q; α is a design parameter.

$$|Z_q| \leq \alpha |A_q| + 1 \tag{2}$$

Intuitively, this means that the number of nodes visited when q is queried is at most one more than some constant proportion of the answer size (the additive term of 1 is required because a node may not be contained in any route ($|A_q| = 0$), but the query procedure must nevertheless visit the queried node to determine this). This guarantees that the amount of work we do for a query is proportional to the output size, provided the output is non-empty, as we can see by combining Equations (1) and (2); for α a constant, this is asymptotically optimal.

$$Q = O((1 + \alpha)|A_q|) \tag{3}$$

Can we build zones so that Inequality 2 is satisfied, without blowing up storage requirements? One bit is required at each node to distinguish boundary nodes from interior nodes, which takes $O(|V|)$ space. The main storage overhead arises from the need to list *all* overlapping routes at a boundary node, even if these routes originate at some other node. This means that a route must be entered into the $L[v]$ list of its bottom node, and of every boundary node between its bottom node and top node. To keep storage requirements in check, our zone construction algorithm builds zones in a bottom-up, greedy way, trying to making zones as large as possible without violating Inequality 2. More precisely, a leaf node is always a boundary node. For a non-leaf node v, we see if v and its children can be placed in the same zone without violating Inequality 2; if not, v is made a boundary node, and otherwise, v is made an interior node. Formalizing this intuitive description, we can define subzones precisely.

Definition 6 *If node v is a leaf node or $(1+\sum_{u\in children(v)}|Z_u|) > (\alpha|A_v|+1)$, then v is a boundary node and Z_v is $\{v\}$. Otherwise, v is an interior node and Z_v is $\{v\} \cup_{u\in children(v)} Z_u$.*

Note that the term $(1 + \sum_{u\in children(v)} |Z_u|)$ is simply the number of nodes that would be visited by a query at node v if v is made an interior node. If this quantity is larger than $(\alpha|A_v| + 1)$, Inequality 2 fails, so we make v a boundary node. Zones are simply maximal subzones; that is, subzones that are not contained within a larger subzone.

3.2.2 Bounding Storage Requirements

The definition of zones lets us bound storage requirements as follows. Denote by X the set of boundary nodes that are not leaves. If $v \in (V - X)$, then only routes whose bottom node is v are listed in $L[v]$. Each route in A appears in the list of its bottom node and, possibly, in the list of some other node in X. For a boundary node v, $|L[v]| = |A_v|$. Hence, we have:

$$\sum_{v\in V}|L[v]| = \sum_{v\in(V-X)}|L[v]| + \sum_{v\in X}|L[v]| \leq |A| + \sum_{v\in X}|A_v|. \tag{4}$$

From Definition 6, if $v \in X$, then

$$|A_v| < \sum_{u\in children(v)} |Z_u|/\alpha. \tag{5}$$

When we sum over $v \in X$ both sides of Inequality (5), we see that the right hand side evaluates at most to $|V|/\alpha$, since all subzones Z_u's involved in the resulting double summation are disjoint. Hence, $\sum_{v\in X}|A_v| \leq |V|/\alpha$, which, used in Equation (4) yields:

$$\sum_{v\in V}|L[v]| \leq |A| + |V|/\alpha. \tag{6}$$

In conclusion, to store $\mathcal{APT}$, we need $O(|V|)$ space for the postdominator tree, $O(|V|)$ further space for the $Bndry?$ bit and for list headers, and finally, from Inequality(6), $O(|A| + |V|/\alpha)$ for the list elements. All together, we have $S = O(|A| + (1 + 1/\alpha)|V|)$, as stated in Theorem 2.

We observe that design parameter α embodies a tradeoff between query time (increasing with α) and preprocessing space (decreasing with α). In fact, for $\alpha < 1/|A|$, we obtain single-node zones (essentially, the control dependence graph since every node has its overlapping routes explicitly listed) and, for $\alpha \geq |V|$, we obtain a single zone (ignoring `START` and `END` and assuming $|A_v| > 0$ for all other nodes, which is the case for the control dependence problem). Small constant values such as $\alpha = 1$ yield a reasonable compromise. Figure 4(c) shows the zone structure of the running example for $\alpha = 1$.

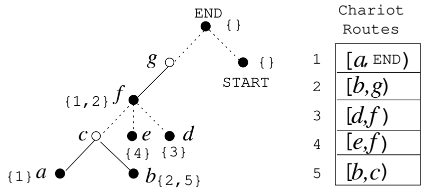
3.2.3 Preprocessing Algorithm

We now describe an algorithm to construct the search structure $\mathcal{APT}$ in linear time. The preprocessing algorithm takes three inputs:

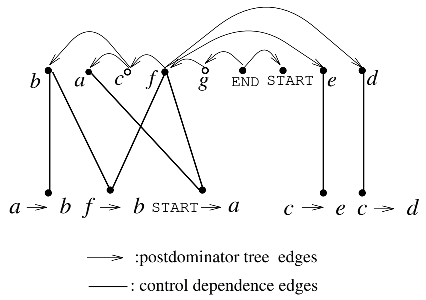
- Tree T for which we assume that the relative order of two nodes one of which is an ancestor of the other can be determined in constant time. For the control dependence problem, this is the postdominator tree.
- The array of routes, A, in which routes are sorted by top endpoint. For the control dependence problem, this array is constructed by Procedure **BuildSortedRoutes** shown in Figure 3.
- Real parameter $\alpha \geq 0$, which controls the space/query-time tradeoff, as described in the previous section.

The preprocessing algorithm consists of a sequence of few simple stages.

1. *For each node v, compute the number of routes whose top node (resp. bottom node) is v.* Let $b[v]$ (resp. $t[v]$) be the number of routes in A with bottom (resp. top) endpoint at v. To compute $b[v]$ and $t[v]$, two counters are set up and initialized to zero. Then, for each route in A, the appropriate counters of its endpoints are incremented. This stage takes time $O(|V| + |A|)$, for the initialization of the $2|V|$ counters, and for constant work done for each of the $|A|$ routes.
2. *Compute, for each node v, the size $|A_v|$ of the answer set A_v.* It is easy to see that $|A_v| = b[v] - t[v] + \sum_{u\in children(v)} |A_u|$. This relation allows us to compute the $|A_v|$ values in bottom-up order, using the values of $b[v]$ and $t[v]$ computed in the previous step, in time $O(|V|)$.
3. *Determine boundary nodes.* The objective of this step is to set, at each node, the value of a boolean variable $Bndry?[v]$ that identifies boundary nodes. Definition 6 can be expressed in terms of subzone size $z[v] = |Z_v|$ as follows.



(a) Caching: $\alpha = 1$



(c) Actual Implementation

Node v	$b[v]$	$t[v]$	$a[v]$	$z[v]$	$NxtBndry[v]$	$Bndry?[v]$	$L[v]$
a	1	0	1	1	f	1	$\{1\}$
b	2	0	2	1	f	1	$\{2,5\}$
c	0	1	2	3	f	0	$\{\}$
d	1	0	1	1	f	1	$\{3\}$
e	1	0	1	1	f	1	$\{4\}$
f	0	2	2	1	END	1	$\{1,2\}$
g	0	1	1	2	END	0	$\{\}$
START	0	0	0	1	END	1	$\{\}$
END	0	1	0	1	$-\infty$	1	$\{\}$

(b) Values Computed During Preprocessing

Figure 7: Construction of $\mathcal{APT}$ Structure

If v is a leaf or $(1 + \sum_{u \in children(v)} z[u]) > (\alpha|A_v| + 1)$, then v is a boundary node, and $z[v]$ is set to 1. Otherwise, v is an interior node, and $z[v] = (1 + \sum_{u \in children(v)} z[u])$.

Again, $z[v]$ and $Bndry?[v]$ are easily computed in bottom-up order, taking time $O(|V|)$.

4. *Determine, for each node v, the next boundary node $NxtBndry[v]$ in the path from v to the root.* If the parent of v is a boundary node, then it is the next boundary for v. Otherwise, v has the same next boundary as its parent. Thus, $NxtBndry[v]$ is easily computed in top-down order, taking $O(|V|)$ time. A special provision is made for the root of T, whose next boundary is set by convention to $-\infty$, considered as a proper ancestor of any node in the tree.

5. *Construct list $L[v]$ for each node v.* By Definition 5, a given route $[b, t)$ appears in list $L[v]$ for $v \in W$, where W contains b as well as all boundary nodes contained by $[b, t)$. Specifically, let $W = \{w_0 = b, w_1, ..., w_k\}$, where $w_i = NxtBndry[w_{i-1}]$, for $i = 1, 2, ..., k$ and w_k is the proper descendant of t such that t is a descendant of $NxtBndry[w_k]$.

   Lists $L[v]$'s are formed by scanning the routes in A in which routes have been entered in decreasing order of top endpoint. Each route ρ is appended at the end of (the constructed portion of) $L[v]$ for each node v in the set W corresponding to ρ. This procedure ensures that, in each list $L[v]$, routes appear in decreasing order of top endpoint.

   This stage takes time proportional to the number of append operations, which is $\sum_{v \in V} |L[v]| = O(|A| + |V|/\alpha)$.

In conclusion, we have shown that the preprocessing time $T = O(|A| + (1 + 1/\alpha)|V|)$, as claimed.

Figure 6 shows the pseudo-code for building the search structure $\mathcal{APT}$. All the preprocessing, including construction of the route array A, can be done in one top-down and one bottom-up walk of the postdominator tree, followed by one traversal of the route array.

3.3 `cdequiv` queries

To solve the `cdequiv` problem efficiently, we exploit an algorithm of Johnson, Pearson and Pingali for identifying tree nodes contained in the same set of routes [JPP94, Section 3]. This algorithm requires $O(|A| + |V|)$ time and space. During preprocessing, we execute this algorithm, and then chain nodes in each equivalence class into a cycle, using a field C at each node, which is made to point to the next node in the cycle. Given a query $\texttt{cdequiv}(v)$, we traverse the cycle associated with node v and output all nodes encountered in this traversal.

```
Procedure PreProcessing(T:tree,A:RouteArray,α:real);
{
  % b[v]/t[v]: number of routes with bottom/top node v
  for each node v in T do
    b[v] := t[v] := 0; od
  for each route [x, y) in A do
    Increment b[x];
    Increment t[y];
  od ;
  %Determine boundary nodes.
  for each node v in T in bottom-up order do
    %Compute output size when v is queried.
    a[v] := b[v] - t[v] + Σ_{u∈children(v)} a[u];
    z[v] := 1 + Σ_{u∈children(v)} z[u]; %Tentative zone size.
    if (v is a leaf) or (z[v] > α * a[v] + 1)
      then % Begin a new zone
        Bndry?[v] := true;
        z[v] := 1;
      else %Put v into same zone as its children
        Bndry?[v] := false;
    endif
  od ;
  % Chain each node to the first boundary node that is
  an ancestor.
  for each node v in T in top-down order do
    if v is root of postdominator tree
      then NxtBndry[v] := - ∞;
      else if Bndry?[parent(v)]
        then NxtBndry[v] := parent(v);
        else NxtBndry[v] := NxtBndry[parent(v)];
        endif
    endif
  od
  % Add each route in A to relevant L[v]
  for i := 1 to |A| do
    let A[i] be [b, t);
    w := b;
    while t is proper ancestor of w do
      append i to end of list L[w];
      w := NxtBndry[w];
    od
}
```

Figure 6: Building the $\mathcal{APT}$ Structure

An interesting point to note is that the algorithm for `cdequiv` in [JPP94] uses the depth-first tree of the undirected version of the control-flow graph, in which the analogs of routes are back edges. This algorithm is based on a non-trivial characterization of `cdequiv` classes in terms of *cycle equivalence*, a relation that holds between two nodes when they belong to the same set of cycles. This characterization, which is remarkable in that it does not make any explicit reference to the postdominance relation, allows the `cdequiv` relation to be computed in less time than it takes to compute the postdominator tree. However, since postdominator information is available in $\mathcal{APT}$, the reduction to cycle equivalence is not needed here.

3.4 Summary

We can summarize the data structure $\mathcal{APT}$ for the Roman Chariots problem as follows.

1. T: tree that permits top-down and bottom-up traversals
2. A: array of chariot routes of the form $[v, w)$ where w is an ancestor of v in T
3. $dfs[v]$: dfs number of node v
4. $Bndry?[v]$: boolean. Set to true if v is a boundary node, and set to false otherwise
5. $L[v]$: list of chariot routes. If v is a boundary node, $L[v]$ is a list of all routes containing v; otherwise, it is a list of all routes whose bottom node is v.
6. $C[v]$: node. Node next to v in `cdequiv` equivalence class of v

Two aspects of our $\mathcal{APT}$ implementation for the control dependence problem are worth mentioning. Rather than use `cd` sets, we work with the corresponding CFG edges, and the conversion to `cd` sets is done on the fly, using the postdominator tree (see Figure 7(c)). This enables the output of `conds` queries to be produced directly without translation from integers to CFG edges, eliminating a data structure that would be needed for this translation. Finally, in procedure **Query** of Figure 5, it is worth inlining the call to procedure **Visit**, and eliminating ancestorship tests on routes cached at the query node itself; if full caching is performed, the overhead of a `conds` query in $\mathcal{APT}$, compared to that in the CDG, reduces to a single conditional test.

4 Experiments

For control dependence investigations, the standard model problem is a nest of repeat-until loops, where the problem size is the number of nested loops, n. Figures 8(a) and (b) show storage requirements as problem size is varied. The storage axis measures the total number of routes stored at all nodes of the tree. The storage required for the CDG is $n(n + 3)$ which grows quadratically with problem size as expected. For a fixed problem size, the storage needed for APT is between the storage needed for the CDG (full caching) and the storage needed if there is no caching (the dotted line at the bottom of Figures 8(a,b)).

Consider the graph for $\alpha = 1/32$ in Figure 8(a). For small problem sizes (between 1 and 31), storage requirements look exactly like those of the CDG. For problem sizes larger than 63, storage requirements grow linearly. In between these two regimes is a transitory region. A similar pattern can be observed in the graph for $\alpha = 1/16$. These results can be explained analytically as follows. From Equation 2, it follows that every node is in a zone by itself if, for all nodes q, $|Z_q| \leq \alpha|A_q| + 1 < 2$. This means that for all nodes q, $|A_q| < 1/\alpha$. If the nesting depth is n, it is easy to verify that the largest value of $|A_q|$ is $(n+1)$. Therefore, if $n < (1/\alpha) - 1$, all nodes are in zones by themselves, which is the case for the CDG. This analysis shows the adaptive nature of the $\mathcal{APT}$ data structure. Intuitively, $1/\alpha$ is a measure of the 'budget' for space — if the problem size is small compared with the budget, the algorithm performs full caching. As problem size increases, full caching becomes more and more expensive, until at some point, zones with more than one node start to appear, and the graph for $\mathcal{APT}$ peels away from the graph for the CDG. A similar analytical interpretation is possible for Figure 8(b) which shows storage requirements for $\alpha > 1$. Finally, Figure 8(c) shows that for a fixed problem size, storage requirements increase as α decreases, as expected. The dashed line is the minimum of the CDG size and the right hand side of Inequality 6 for $n = 100$; this is the computed upper bound on storage requirements, and it clearly lies above the graph of storage actually used.

Figure 8(d) shows that for a fixed problem size, worst case query time decreases as α decreases. Because actual query time is too small to measure accurately, we measured instead the number of routes examined during querying (say r), and the number of nodes in the subzone of the query node, other than the query node itself (say s). The y-axis is the sum $(r + 2s)$, where the factor of 2 comes from the need to traverse each edge in the subzone twice, once on the way down and then again on the way back up. Note that each graph levels off at its two ends (for very small α and for very large α) as it should. It is important to note that the node for which worst-case query time is exhibited is different for different values of α. In other words, the range of query times for a fixed node is far more than the 5:1 ratio seen in Figure 8(d).

Finally, Figures 8(e,f) show how preprocessing time varies with problem size, and with α. These times were measured on a SUN-4. Note that for $\alpha > 1/8$, preprocessing time is less than the time to build the postdominator tree; even for very small values of α, the time to build the $\mathcal{APT}$ data structure is no more than twice the time to build the postdominator tree. This shows that preprocessing is relatively inexpensive.

'Real' programs, such as the SPEC benchmarks, are less challenging than the model problem. Note that the ratio of storage required for full caching to the storage required for no caching is simply the average number of nodes in a chariot route. In terms of the control dependence relation, this is the average number of nodes that are control dependent on a CFG edge, considering only CFG edges with non-empty `cd` sets. For the model problem of nested repeat-until loops, this number grows with problem size. For procedures in the SPEC benchmarks, this number is fairly independent of problem size, and is close to 2, as can be seen in Figure 9(a). This is because the height of the postdominator tree of most real programs is quite small, and it is more or less independent of program size (for example, in the SPEC benchmarks, most procedures have a postdominator tree height less than 25, and only procedure *iniset.f* in *doduc* has a postdominator tree height of more than 75). Therefore, in practice, the size of the CDG grows linearly with program size, as is seen in Figure 9(a). We conclude that the $\mathcal{APT}$ data structure can reduce storage requirements for caching by about a factor of half, and this can be seen in Figures 9(a,b). Query time was not significantly affected when α was set to 1; for larger values of α, query time for a few nodes was affected, but on the whole, the effect was small. Finally, for every procedure in the SPEC benchmarks, preprocessing time to construct $\mathcal{APT}$ is a small fraction of the time to build the postdominator tree.

5 Applications

The $\mathcal{APT}$ data structure can be used to design an optimal algorithm for finding the *dominance frontier* of a node, which is the key step in conversion to Static Single Assignment (SSA) form [SS70, CFR+91]. In conversion to SSA form, dummy assignments called ϕ-functions are introduced into the program in such a way that every use of a variable is reached by at most one real or dummy assignment. The $\mathcal{APT}$ data structure can be used to convert a program to SSA form in $O(|E|)$ time per variable; the related problem of computing sparse dataflow evaluator graphs can also be solved using the same idea.

First, we review the standard definition of dominance.

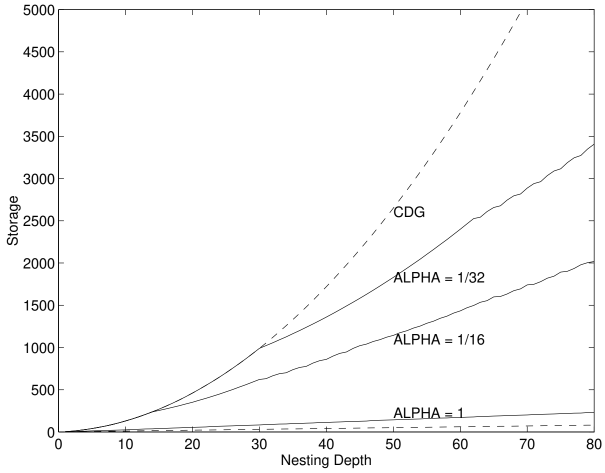
Definition 7 *A node w is said to* **dominate** *a node v if every path from* `START` *to v contains w.*

It is well-known that any algorithm computing the postdominance relation can be used to compute the dominance relation by applying this algorithm to the *reverse control flow graph*, the control flow graph obtained by reversing all control flow edges and interchanging `START` and `END`[LT79].

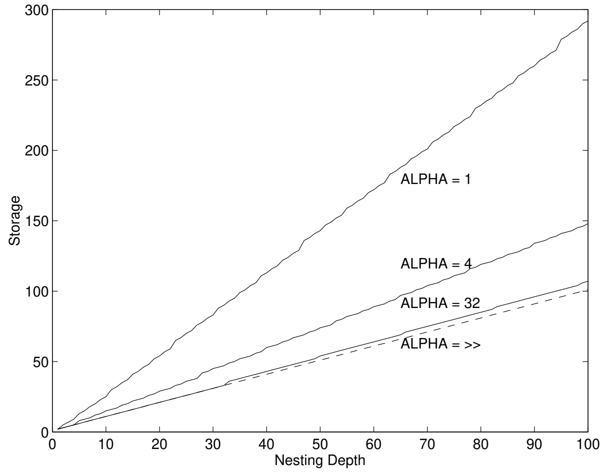
Definition 8 *The* **edge dominance frontier** *of a node w in a control flow graph is a set of edges, denoted by* $\texttt{edf}(w)$*, such* $(v \to u) \in \texttt{edf}(w)$ *if*

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

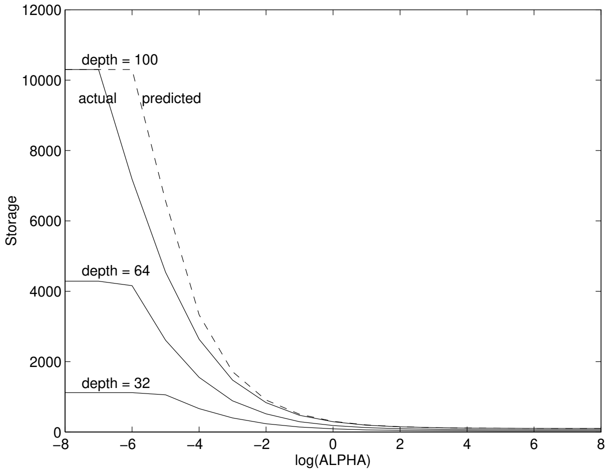
This definition is similar to that of Cytron *et al* [CFR+91]. Comparing this to Definition 3, it is clear that $(v \to u)$



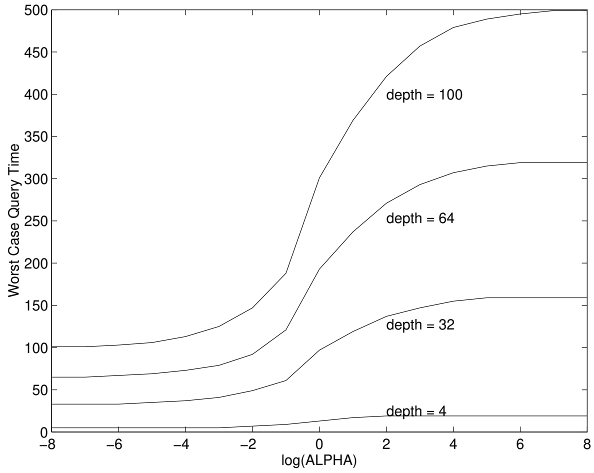
(a)Storage vs. Nesting Level ($\alpha \leq 1$)



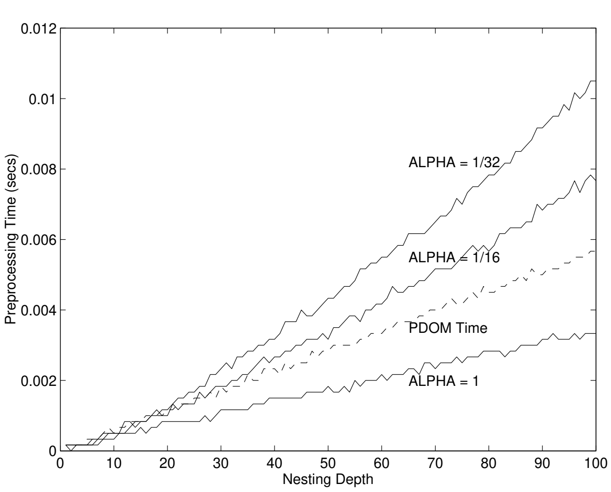
(b) Storage vs. Nesting Level ($\alpha \geq 1$)



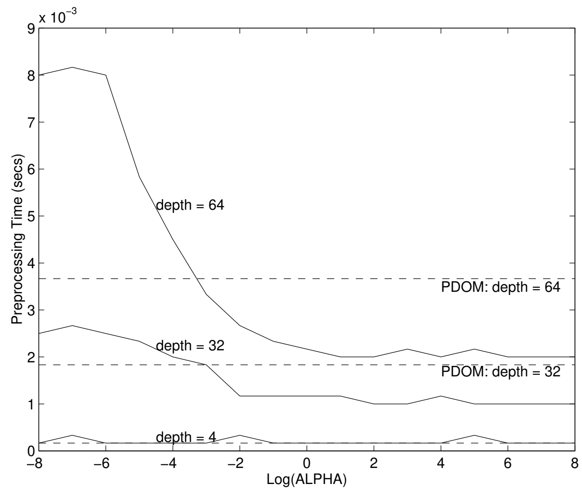
(c) Storage vs. α



(d) Query Time vs. α



(e) Preprocessing Time vs. Nesting Level



(f) Preprocessing Time vs. α

Figure 8: Experimental Results for Repeat-Until Loop Nests

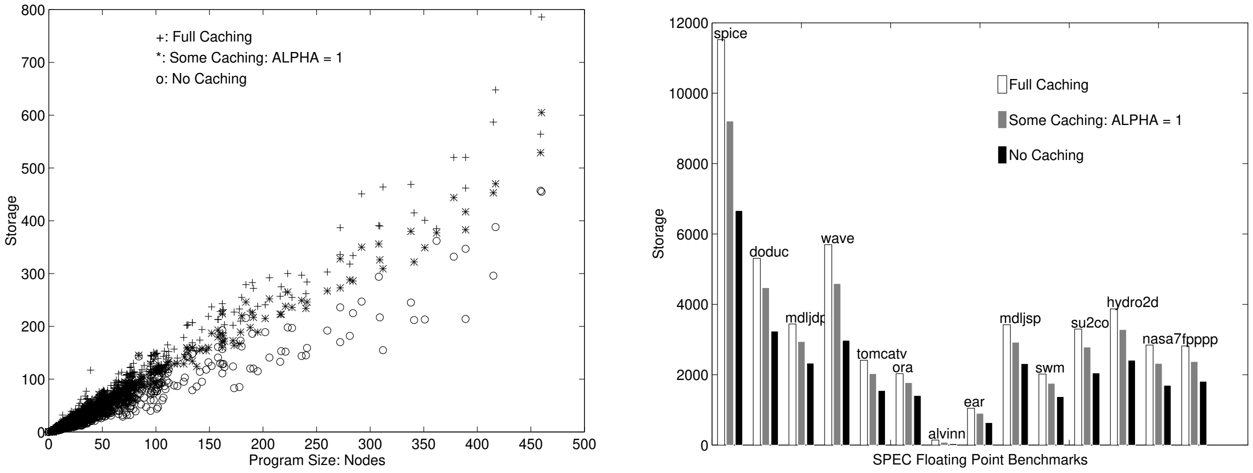


Figure 9: Experimental Results for SPEC Benchmarks

belongs to $\texttt{edf}(w)$ iff $(u \rightarrow v)$ belongs to $\texttt{conds}(w)$ in the reverse control flow graph. Therefore, an optimal data structure for the dominance frontier problem is obtained by constructing an $\mathcal{APT}$ data structure on the reverse control flow graph (this data structure can be viewed as an *augmented dominator tree* on the forward control flow graph). Given a query for $\texttt{edf}(w)$, node w is queried in the data structure using the query procedure of Figure 5, and each edge produced by the query procedure is reversed before being output.

To find the SSA form of a program, a set of nodes S containing assignments to a variable is given, and it is desired to find a set T such that T is closed in the following sense: if node t belongs to T, and $(v \rightarrow u)$ belongs to $\texttt{edf}(t)$, then u belongs to t. This is called an *iterated* dominance frontier computation, and dummy assignments called ϕ-functions must be introduced at nodes in T which are not in S. In the reverse control flow graph, this computation can be described equationally as follows. Extend the definition of conds to *sets* of nodes in the natural way: if S is a set of nodes, let $\texttt{conds}(S)$ be the set $\bigcup_{s \in S} \texttt{conds}(s)$. Let source be a function that, given an edge, returns the source of the edge; this function too can be extended to sets of edges in the natural way. The ϕ-function placement problem can be reduced to the following problem on the reverse CFG: given a set of nodes S, compute the smallest set of nodes T such that $T = S \cup \texttt{source}(\texttt{conds}(T))$.

First, consider the problem of using the $\mathcal{APT}$ data structure to compute $\texttt{conds}(N)$ where N is a set of nodes known before any queries are made. To avoid visiting tree nodes repeatedly during querying, it is desirable to sort the nodes in N by level in the tree, and query the nodes of N in bottom-up order. A node in the tree is marked when it is queried, and the query procedure of Figure 5 is modified so that it never visits nodes below a marked node. Therefore, the time for querying all the nodes in N is proportional to the sum of the number of nodes visited and the number of paths in the L_v lists of these nodes. In the worst case, every node in the tree is visited, in which case, using Equation 6, we can see that the total time for querying is

$$Q = O(|V| + |A| + |V|/\alpha). \tag{7}$$

Since $|A| \leq |E|$ in our context, this bound can be written as $O(|E| + (1 + 1/\alpha)|V|)$, which, for constant α, is proportional to program size. Sorting the set N initially by level can be done by a breadth-first walk of the tree, so it can be done in time $O(|V|)$. Therefore, we can find the set $\texttt{conds}(N)$ for a set of nodes N in time proportional to program size.

If N is given online, we cannot sort it before starting queries. However, as long as nodes are presented for querying in order of decreasing level number, the approach discussed above can be used. To accomplish this in our context, we maintain a priority queue [CLR92] of nodes that must be queried; the key for the priority queue is level number in the tree. Initially, this priority queue contains only the nodes in S. At each step, a node w of highest level (farthest from the root) is extracted from the priority queue, and a query is made in the $\mathcal{APT}$ data structure with that node; if there is a node u in $\texttt{source}(\texttt{conds}(w))$ but which is not in the output set, it is added to the output set, and then inserted into the priority queue. From Theorem 1, it follows that the level of u is less than or equal to the level of w. Therefore, querying a node at level l can never cause the insertion of a node at a level strictly greater than l into the priority queue.

The priority queue can be implemented using a heap, which gives $O(log(k))$ time per operation, where k, the number of keys, is the height of the tree [CLR92]; a more sophisticated data structure due to van Emde Boas *et al* gives $O(log(log(k))$ time per operation [VEBKZ77]. Priority queues are more general than what we need since we can guarantee that insertions always occur 'behind' extractions. Sreedhar and Gao have pointed out that an array of size k suffices [SG95]; in this array, each element is a linked list of nodes, at the corresponding level, that must be queried. A node at level l is inserted into the priority queue by appending it to the list of nodes at array element l; this gives $O(1)$ time for each insertion. To extract a node with maximum level number, the array elements are scanned by decreasing level number till a non-*nil* linked list is found; this gives $O(k)$ time for extractions. However, since insertions always occur behind extractions, the extraction process always advances monotonically through the array towards decreasing level number, so the total time for inserting and extracting nodes during the computation is bounded by $O(|V|)$.

The algorithm is described in Figure 10. The input to Procedure ϕ-placement is the set of nodes containing real assignments to some variable; the output is the set of nodes that should contain real or dummy assignments to the variable. The running time of the algorithm is proportional to the size of the program. This is an improvement over the original algorithm of Cytron *et al* which has a worst-case time complexity that is quadratic in the size of the program [CFR+91]. Cytron and Ferrante have used path compression to design another algorithm with $O(|E|\alpha(|E|))$ complexity [CF93], but this algorithm is rather complicated. Johnson and Pingali have designed an $O(|E|)$ algorithm for ϕ-function placement [JP93], using the dependence flow graph [PBJ+91]. Just as building the SSA form can be reduced to an iterated `conds` computation, building the dependence flow graph can be reduced to an iterated `cdequiv` computation, and can be done in $O(|E|)$ time per variable. However, this approach cannot be extended to compute sparse dataflow evaluator graphs. Recently, Sreedhar and Gao have used a data structure called the *DJ-graph* to design a fast algorithm that takes $O(|E|)$ time for ϕ-function placement, and for computing sparse dataflow evaluator graphs. Their algorithm uses an $O(|E|)$ query procedure for finding `conds` sets. In contrast, our approach requires a small amount of preprocessing to build the $\mathcal{APT}$ data structure, the pay-off being proportional time access to `conds` sets. The algorithm of Cytron *et al* can be viewed as one extreme of our algorithm, when there is full caching of dominance frontiers; similarly, the Sreedhar and Gao algorithm can be viewed as the other extreme of our algorithm, when no caching is performed. One final remark is in order — although the running time of the algorithm in Figure 10 is $O(|E|)$, the algorithm may not be optimal since its running time is not necessarily proportional to the size of the *output*. Is there an optimal algorithm for SSA construction?

```
Procedure φ-placement (N: set of nodes);
{
    % APT data structure on reverse CFG is global
    Create a Priority Queue PQ;
    Insert nodes in set N into PQ;
    In tree T, mark all nodes belonging to set N;

    while PQ is not empty do
      q := ExtractMax(PQ);
      Output q;
      QueryIncr(q);
    od ;
    Delete marks from nodes in T;
}
Procedure QueryIncr(QueryNode);
{
    VisitIncr(QueryNode, QueryNode);
}
Procedure VisitIncr(QueryNode,VisitNode);
{
    for each route i in L[VisitNode]
      in list order do
        let A[i] be [b, t);
        let reverse CFG edge corresponding to
          Route i be u → b;
        if t is strict ancestor of QueryNode
        then if u is not marked
              then
                Mark u;
                Insert u into PQ;
              endif ;
        else break ; % exit from the loop
      od ;
    if VisitNode is not a boundary node
      then
        for each child C of VisitNode
          do
            if C is not marked
              then VisitIncr(QueryNode,C);
          od ;
    endif ;
}
```

Figure 10: Computing Iterated Dominance Frontiers

6 Conclusions and Related Work

Control dependence was first defined by Ferrante, Ottenstein and Warren [FOW87]. They also described the control dependence graph, and gave an optimal algorithm for `cd` queries, which used the postdominator tree to enumerate `cd` sets in proportional time. Cytron, Ferrante and Sarkar described quadratic time algorithms for `conds` and cdequiv queries [CFS90]. Sreedhar and Gao investigated the conds problem using their DJ-graph representation, but this approach did not reduce the asymptotic complexity of conds computation [SGL94]. The cdequiv problem for *reducible* control flow graphs was solved by Ball [Bal93] who needed both dominator and postdominator information in his solution; subsequently, Podgurski gave a linear-time algorithm for *forward* control dependence equivalence, which is a special case of general control dependence equivalence [Pod93]. The general cdequiv problem was solved finally by Johnson, Pearson and Pingali who designed an optimal algorithm which required $O(|E|)$ preprocessing time and space, and which enumerated cdequiv sets in proportional time [JPP94]. This algorithm requires neither dominator nor postdominator information, and permits the computation of the cdequiv relation in less time than it takes to compute the postdominance relation!

There are many alternatives to the zone construction algorithm given here. For example, instead of searching the subtree below a query node for the bottom ends of chariot routes, we can search the path from the query node to END for the top ends of relevant chariot routes. In general, there is a trade-off between the sophistication of the query procedure and the amount of caching in $\mathcal{APT}$, for a given query time. For example, we can use cdequiv information in answering conds queries. It can be shown that the nodes in a cdequiv equivalence class are ordered by the ancestor relation in the postdominator tree [JPP94]. Given a query conds(v), we can answer instead the query conds(w) where w is the node in the cdequiv class of v that is lowest is the tree; this lets the query procedure avoid examining nodes on the path $[v, w)$, which can be exploited during zone construction to reduce caching.

Although we have used the term *caching* to describe $\mathcal{APT}$, note that most caching techniques for search problems, such as memoization and related ideas used in the theorem proving community [Mic68, SS93], perform caching at *run time* (query time). In contrast, caching in $\mathcal{APT}$ is performed during preprocessing, and the data structure is not modified by query processing. This permits us to get a grip on storage requirements, which is difficult to do with run time approaches. Finally, we note that there is a deep connection between $\mathcal{APT}$, and the use of factoring to reduce the size of the CDG [CFS90]. Factoring identifies nodes that have control dependences in common, and creates representations which permit control dependences to be shared by multiple nodes. The simplest kind of factoring exploits cdequiv sets. If p nodes are in a cdequiv set, and have q routes in common, we can introduce a *junction* node, connect the q routes to the junction, and introduce edges from the junction to each of the p nodes. In this way, the number of edges in the data structure is reduced from $p * q$ to $p + q$. Exploitation of cdequiv information alone is not adequate to reduce the asymptotic size of the graph, but the idea of sharing routes can be extended — for example, factoring is possible when the routes containing a node v_1 are a subset of the routes containing node v_2. However, no factorization to date has reduced worst-case space requirements. To place the $\mathcal{APT}$ data structure in perspective, note that it can be viewed as a factored representation since a route is cached just once per zone, and that entry is shared by all nodes in the zone. However, there is an important difference between the traditional approaches to factorization, and the one that we have adopted in $\mathcal{APT}$. In previous factorizations, *every* route encountered during query processing is reported as output. In our approach, the query procedure may encounter some irrelevant routes which must be 'filtered out', but there is a guarantee that the number of irrelevant routes encountered during query processing is at most some constant fraction of the actual output. By permitting this slack in the query procedure, we are successful in reducing space and preprocessing time requirements without affecting asymptotic query time.

More generally, the approach to conds described in this paper can be viewed as an example of Chazelle's *filtered search* [Cha86], a technique used in computational geometry to solve *range search* problems. In these problems, a set of geometrical objects in R^d is given. A query is made in the form a connected region in R^d, and all objects intersecting this region must be enumerated. To draw the analogy, we can view the routes in our problem as geometric objects, and we can view the query node as the analog of the query region; clearly, the conds problem asks for enumeration of all 'objects' that intersect the query 'range'. Filtered search exploits the fact that to report k objects, it takes $\Omega(k)$ time. Therefore, we can invest $O(k)$ time in an adaptive search technique that is relatively less efficient for large k than it is for small k. In our solution to the conds problem, nodes contained in a large number of routes are allowed to be in zones with a large number of nodes; therefore, a query at such a node may visit a large number of nodes, but this overhead is amortized over the size of the output. Correspondingly, the search procedure visits a small number of nodes if the query node has only a small amount of output. This kind of search procedure with adaptive caching may prove useful in solving other problems in the context of restructuring compilers.

Acknowledgments: We thank Richard Johnson and V. Sreedhar for their implementations, and for feedback. Richard also played a major role in designing the control dependence equivalence algorithm reported in PLDI '94. Paul Chew, Mayan Moudgill, Michael Wolfe, Eric Stoltz, Richard Schooler and Ruth Pingali caught errors in drafts of this paper.

References

[ABC+88] Frances Allen, Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, and Vivek 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.

[Bal93] Thomas Ball. What's in a region? or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems*, 2(1–4):1–16, March–December 1993.

[BR91] David Bernstein and Michael 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.

[CF93] Ron Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, pages 461–476, August 1993. Published as Lecture Notes in Computer Science, number 768.

[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.

[Cha86] Bernard Chazelle. Filtering search: A new approach to query answering. *SIAM Journal of Computing*, 15:703–724, 1986.

[CicBC] Marcus Tullius Cicero. *Pro L. Cornelio Balbo Oratio 39*. Published by Senate of Rome, Rome, 56 BC.

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

[Fis81] Josh Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 7(3):478–490, 1981.

[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.

[GPS90] Rajiv Gupta, Lori Pollock, and Mary Lou Soffa. Parallelizing data flow analysis. In *Proceedings of the Workshop on Parallel Compilation*, Kingston, Ontario, May 6–8, 1990. Queen's University.

[GS87] Rajiv Gupta and Mary Lou Soffa. Region scheduling. In *2nd International Conference on Supercomputing*, pages 141–148, 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.

[HPR87] Susan Horowitz, Jan Prins, and Thomas 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, January 21–23, 1987.

[Joh94] Richard Johnson. *Efficient Program Analysis using Dependence Flow Graphs*. PhD thesis, Cornell University, August 1994.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, Albuquerque, New Mexico, June 23–25, 1993.

[JPP94] Richard Johnson, David Pearson, and Keshav 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.

[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.

[Mic68] D. Michie. 'Memo' functions and machine learning. *Nature*, 218:19–22, April 1968.

[PBJ$^+$91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence Flow Graphs: An algebraic approach to program dependencies. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 67–78, January 1991.

[Pod93] Andy Podgurski. Reordering-transformations that preserve control dependence. Technical Report CES-93-16, Case Western Reserve University, July 1993.

[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.

[SGL94] Vugranam C. Sreedhar, Guang R. Gao, and Yong-fong Lee. DJ-graphs and their applications to flowgraph analyses. Technical Report ACAPS Memo 70, McGill University, May 1994.

[SS70] R. M. Shapiro and H. Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, February 1970.

[SS93] A. Segre and D. Scharstein. Bounded-overhead caching for definite-clause theorem proving. *Journal of Automated Reasoning*, 11:83–113, August 1993.

[VEBKZ77] P. Van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.