Optimal Control Dependence Computation and the Roman Chariots Problem

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

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

August 7, 1996

Abstract

The *control dependence* relation plays a fundamental role in program restructuring and optimization. The usual representation of this relation is the *control dependence graph* (CDG), but the size of the CDG can grow quadratically with the input program, even for structured programs.

In this paper, we introduce the *augmented postdominator tree* ($\mathcal{APT}$), a data structure which can be constructed in space and time proportional to the size of the program, and which supports enumeration of a number of useful control-dependence sets in time proportional to their size. Therefore, $\mathcal{APT}$ provides an *optimal representation of control dependence.*

Specifically, the $\mathcal{APT}$ data structure supports enumeration of the set $\mathtt{cd}(e)$, which is the set of statements control dependent on control-flow edge e, of the set $\mathtt{conds}(w)$, which is the set of edges on which statement w is dependent, and of the set $\mathtt{cdequiv}(w)$, which is the set of statements having the same control dependences as w.

Technically, $\mathcal{APT}$ can be viewed as a factored representation of the CDG where queries are processed using an approach known as *filtered search.*

[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.

1 Introduction

Control dependence is a key concept in program optimization and parallelization. Intuitively, a statement w is control dependent on a statement u if u is a conditional that affects the execution of w. For example, in an *if-then-else* construct, statements on the two sides of the conditional statement are control dependent on the predicate. In the presence of nested control structures, multiway branches and unstructured flow of control, intuition is an unreliable guide, and one needs to rely on a formal, graph-theoretic definition of control dependence, based on the following concepts.

Definition 1 *A* **control flow graph** $G = (V, E)$ *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.*

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.

Any node v is 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*

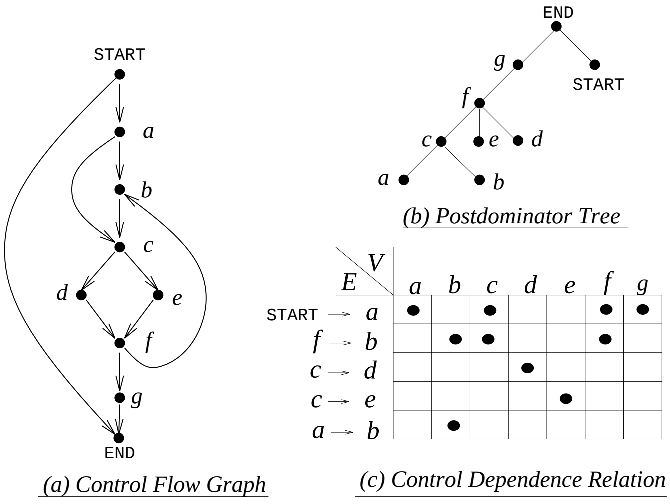
1. w *postdominates* v*, and*
2. *if* $w \neq u$*, then* w *does not postdominate* u.

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.

Definition 4 *Given a control flow graph* $G = (V, E)$*, its* **control dependence relation** *is the set* $C \subseteq E \times V$ *of all pairs* (e, w) *such that node* w *is control dependent on edge* e.

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]:

Definition 5 *Given a node* w *and an edge* e *in a control program graph with control dependence relation* C*, we define the following* **control dependence sets***:*



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

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

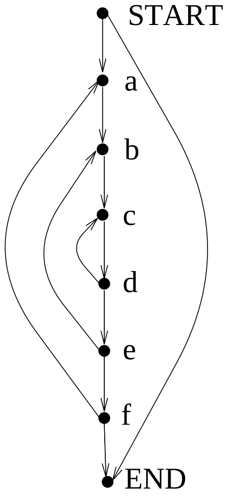
Set $\texttt{cd}(e)$ is the set of nodes that are control dependent on edge e, while $\texttt{conds}(w)$ is the set of control dependences of node w. These sets are used in scheduling instructions across basic block boundaries for speculative or predicated execution [Fis81, BR91, NNS94], and in merging program versions [HPR87]. They are also useful in automatic parallelization [FOW87, ABC$^+$88, SAF90]. 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 answer `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 $\Omega(|V||E|)$,



E \ V	a	b	c	d	e	f
START -> a	•	•	•	•	•	•
f -> a	•	•	•	•	•	•
e -> b		•	•	•	•	
d -> c			•	•		

Figure 2: A Program with Three Nested Repeat-until Loops

even if we restrict our attention to structured programs. Figure 2 shows a program with three nested repeat-until loops, and its control dependence relation. It can be verified that for programs consisting of n nested repeat-until loops, $|E| = 3n + 2$ and $|C| = n(n + 3)$; therefore, the size of the control dependence relation can grow quadratically with program size even for structured programs.

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 constant time access to the immediate postdominator of a node, as well as proportional time access to all 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 postdominance 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 the bipartite graph $(V, E; C)$. That is, the two sets of nodes in the bipartite graph are V and E, and there is an edge between v and e if v is control dependent on edge e. Since the size of the CDG is $\Omega(|C|)$, which can be $\Omega(|E||V|)$, many attempts have been made to construct more compact representations of C [FOW87, CFS90, Bal93, JP93, SGL94]. 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 *Augmented 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 `cd`, `conds` and `cdequiv` 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 $1/|E|$.

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. The $\mathcal{APT}$ data structure is described incrementally by considering the requirements of the three kinds of control dependence queries. In Sections 3, 4, and 5, we examine `cd`, `conds`, and `cdequiv` queries respectively, and develop the machinery to answer these queries optimally. Experimental results using the SPEC benchmarks are reported in Section 6. Finally, in Section 7, 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 show that the computation of control dependence sets (Definition 5) has a natural graph-theoretic formulation which we call the *Roman Chariots* problem. This formulation exploits 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 6 *Let $T=<V,F>$ be a tree. For $v,w \in V$, let $[v,w]$ denote the set of vertices on the simple path joining v and w in T, and let $[v,w)$ denote $[v,w]-\{w\}$. (In particular, $[v,v)$ is empty.)*

For example, in the postdominator tree of Figure 1(b), $[d,a]=\{d,f,c,a\}$, while $[d,a)=\{d,f,c\}$. This notation is similar to the standard one for open and closed intervals of the line. The following key theorem, due to Ferrante, Ottenstein and Warren [FOW87], shows how edges of the control flow graph are constrained with respect to the postdominator tree and provides a simple characterization of `cd` sets.

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. *$\mathtt{cd}(u \to v)=[v, parent(u))$.*

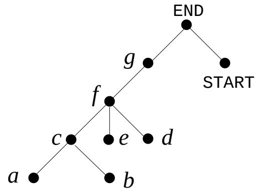
Proof: Note that since no control-flow edge emanates from `END`, the expression $parent(u)$ is defined whenever $(u \to v) \in E$.

1. If $parent(u)$ does not postdominate v, we can find a path $v \to \ldots \to$ `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.

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

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 → a	[a, END)
f → b	[b,g)
c → d	[d,f)
c → e	[e,f)
a → b	[b,c)

(b) Postdominator Tree and cd sets

Figure 3: Compact Representation of Control Dependence

2. We show that $\mathtt{cd}(u \to v) \subseteq [v, parent(u))$. Let w be an element of $\mathtt{cd}(u \to v)$. From the definition of control dependence, w must postdominate v, so w is on the path $[v, \mathtt{END}]$ in the postdominator tree. From part (1), $parent(u)$ is also on the path $[v, \mathtt{END}]$. However, w cannot be on the path $[parent(u), \mathtt{END}]$ 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 3 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

[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. This conversion is not needed in this paper.

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.

For the purpose of exposition, it is convenient to assume that the array of `cd` sets, which is indexed by CFG edges in Figure 3, 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 form a tree rooted at Rome, in which nodes represent cities and edges represent roads* [5]*. Public transportation is provided by chariots that travel between a city and one of its ancestors in the tree.*

Given a rooted tree $T =< V, F, ROME >$ *and an array* $A[1..m]$ *of* chariot routes *each specified in the form* $[v, p)$*, where* p *is an ancestor of* v *in* T*, design a data structure that permits enumeration of the following sets.*

1. $\texttt{cd}(\rho)$*: the* cities *on route* ρ*.*
2. $\texttt{conds}(w)$*: the* routes *that serve city* w*.*
3. $\texttt{cdequiv}(w)$*: the* cities *that are served by all and only the routes that serve city* w*.*

For future reference, we introduce the following definition.

Definition 7 *The set of chariots serving a node* v *is a subset of* A*, the set of all chariots, and will be referred to as set* A_v*.*

The control dependence problem is reduced to the Roman Chariots problem as follows. Procedure **ConstructRomanChariots** in Figure 4 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.

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

The correctness of this reduction follows immediately from Theorem 1 and Procedure **ConstructRomanChariots**. In the construction of Figure 4, 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 consider `conds` queries in Section 4. Note that

[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.

```
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 p in T in top-down order do
      for each child u of p do
        for each edge (u → v) in G do
          if v is not p
            then %append a cd set to end of A
            i := i + 1;
            A[i] := [v, p);
            Note the correspondence between
            edge u → v and index i;
          endif
        od
      od
      return T, A;
}
```

Figure 4: Constructing a Roman Chariots Problem

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.

In subsequent sections, we develop a data structure for the Roman Chariot problem, obtained by a suitable augmentation of the given tree T. Motivated by the application to control dependence, we call this data structure an *Augmented Postdominator Tree*, denoted $\mathcal{APT}$. The rest of the paper establishes the following result.

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

- $\texttt{cd}(e)$*: time $O(|\texttt{cd}(e)|)$, independent of α.*
- $\texttt{conds}(w)$*: time $O((1+\alpha)|\texttt{conds}(w)|)$.*
- $\texttt{cdequiv}(w)$*: time $O(|\texttt{cdequiv}(w)|)$, independent of α.*

For the special case when T is the postdominator tree of a control flow graph, we have that $|A| \leq |E|$ and $|V| \leq |E| + 1$, leading to 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 $\mathcal{APT}$: `cd` Queries

No preprocessing is required to answer `cd` queries optimally. 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 that of Ferrante *et al* [FOW87].

4 $\mathcal{APT}$: `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 algorithm 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 5(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 `conds(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 5(b). We call this approach *full caching*, to contrast it to the previous scheme which we call *no caching*. 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.

It is possible to compromise between these two extremes. Suppose we partition the nodes in V into two disjoint sets called *boundary* nodes and *interior* nodes. Although this partition can be made arbitrarily, it is simpler to make all leaf nodes boundary nodes; for now, non-leaf nodes can be classified arbitrarily as boundary or interior nodes. With each node $v \in V$, we associate a list of routes $L[v]$, defined formally as follows.

Definition 8 *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.

In Figure 5, boundary nodes are shown as solid dots, while interior nodes are shown as hollow dots. Figure 5(a) shows one extreme in which all non-leaf nodes are interior nodes, while Figure 5(b) shows the other extreme when all nodes are boundary nodes. In Figure 5(c), nodes c and g are interior nodes, while all other nodes are boundary nodes.

Our `conds` query procedure visits nodes in the subtree below the query node as before, but it exploits boundary nodes to limit the portion of this subtree that it visits. Suppose that the query node is q, and that the query procedure encounters a boundary node x. It is easy to show that the query procedure does not need to visit nodes that are proper descendants of x — any route ρ which contains q and whose bottom node is a proper descendant of x must also contain x; from Definition 8, ρ must be stored at x. Therefore, to answer the query `conds`(q), it is unnecessary to examine the subtree below x since all the relevant chariot routes from this subtree are stored at x itself. For example, in Figure 5(c), when answering the query `conds`(g), it is unnecessary to look below boundary node f, and the query can be answered just by visiting nodes f and g. One way to visualize this is to imagine that the edges connecting a boundary node to its children are deleted from the tree (these edges are never traversed by the query procedure). This leaves a forest of small trees, and the query procedure needs to visit only the descendants of a query node in this forest. We will call each tree in this forest a *zone*; the portion of the forest below a node q will be called the *subzone* associated with node q. These concepts are defined formally as follows.

Definition 9 *A node w is said to be in the* subzone *associated with a node q, referred to as Z_q, if (i) w is a descendant of q, and (ii) the path $[q,w)$ does not contain any boundary nodes.*

A zone *is a maximal subzone; that is, a subzone that is not strictly contained in any other subzone.*

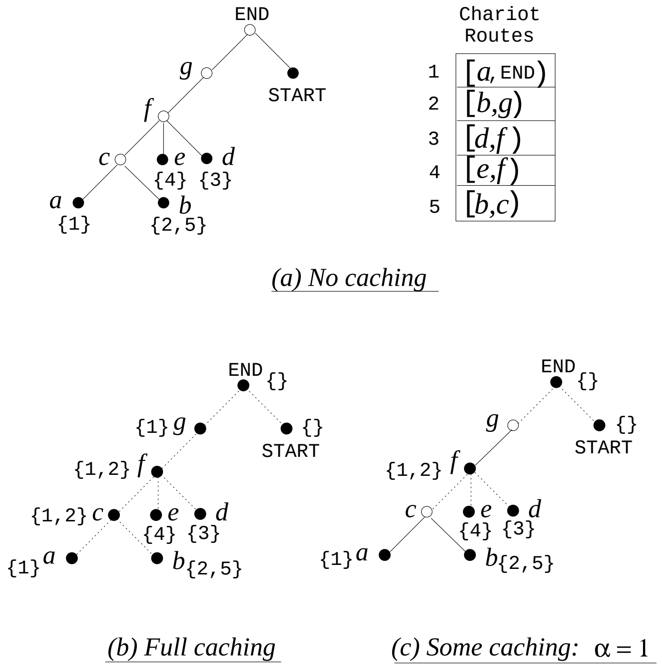


Figure 5: Zone Structure

In Figure 5(c), there are six zones induced by the following sets of nodes: $\{a, b, c\}$, $\{d\}$, $\{e\}$, $\{f, g\}$, $\{\texttt{START}\}$ and $\{\texttt{END}\}$. The subzone associated with node g is the set of nodes $\{f, g\}$. Note that even though Chariot Route 1 contains nodes $\{a, c, f, g\}$, it is stored only at nodes a and f since these are the only boundary nodes it contains.

Given a query node q, the query procedure examines routes stored at nodes in subzone Z_q. 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. A simple implementation of this query procedure is given in Figure 6. 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; an algorithm for determining which nodes are boundary nodes will be described in Section 4.1. In line 4 of Procedure **CondsVisit**, 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.

It follows immediately that the query time is proportional to the sum of the number of visited nodes and the number of reported routes.

```
Procedure CondsQuery(QueryNode);
{
    % APT data structure is global variable;
    % Query outputs list of routes numbers
    CondsVisit(QueryNode, QueryNode);
}
Procedure CondsVisit(QueryNode, VisitNode);
{
    for each route i in L[VisitNode]
      in list order do
        let A(i) be [b, t);
        if t is a proper ancestor of QueryNode
        then output i;
        else break ; % exit from the loop
      od ;
    if VisitNode is not a boundary node
      then
        for each child C of VisitNode
          do
            CondsVisit(QueryNode,C)
          od ;
    endif ;
}
```

Figure 6: Query Procedure for `conds`

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

Next, we discuss how zones can be constructed to obtain optimal query time without blowing up space requirements.

4.1 Criterion for Zones

To obtain optimal query time, we require that the following inequality hold for all nodes q; α, a non-negative real number is a design parameter.

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

Intuitively, 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 prevents zone Z_q from becoming empty when q is not contained in any route ($|A_q| = 0$). By combining Equations (1) and (2), we see that

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

Thus, the amount of work done for a query is basically proportional to the output size; for α a constant, this is asymptotically optimal.

To get some intuition for the significance of α, consider what happens if we fix the problem and vary α. If α is set to 0, the size of the subzone associated with each node is 1. This means that each node is in a zone by itself, which corresponds to full caching. At the other extreme, if we choose a very large value of α, nodes can be contained in arbitrarily large zones, and the situation corresponds to no caching. Thus, by varying α, we get the full range of behavior from full caching to no caching.

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.

Our zone construction algorithm is a simple bottom-up, greedy algorithm that tries 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 all 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[6]. Formalizing this intuitive description, we obtain a definition for subzone construction.

Definition 10 **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* $\{v\}$. **Else**, 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 were 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.

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 10, 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

[6] A variation of this scheme is to allow a node to be in the same zone as some but not all of its children. We do not need this complication, but it may be possible to exploit this idea to reduce storage requirements further.

summation are disjoint. Hence, $\sum_{v \in X} |A_v| \leq |V|/\alpha$, which, used in Relation (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 5(c) shows the zone structure of the running example for $\alpha = 1$.

4.2 Preprocessing for `conds` Computations

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 **ConstructRomanChariots** shown in Figure 4.

- Real parameter $\alpha \geq 0$, which controls the space/query-time tradeoff, as described in the previous section.

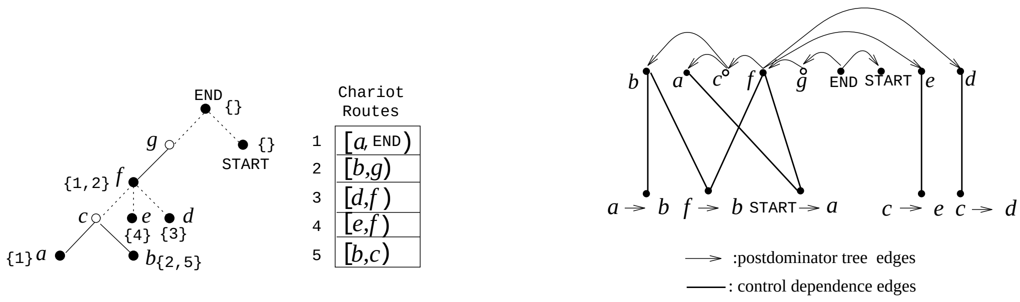
The preprocessing algorithm consists of a sequence of few simple stages.

1. *For each node v, compute the number of routes whose bottom (resp., top) 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 $O(|V|)$.

```
Procedure CondsPreprocessing(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 7: Constructing the $\mathcal{APT}$ Structure



(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 8: The $\mathcal{APT}$ Structure and Its Parameters

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 10 can be expressed in terms of subzone size $z[v] = |Z_v|$ as follows.

   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 8, 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 7 shows the pseudo-code for building the search structure. 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.

5 $\mathcal{APT}$: `cdequiv` Queries

The routes in a Roman Chariots problem induce a natural equivalence relation on cities: two cities are placed in the same equivalence class if and only if they are served by the same set of routes. In this section, we describe an algorithm that constructs a representation of this equivalence relation during the preprocessing phase. This representation uses *region* nodes [FOW87] to represent each `cdequiv` class; if a city v is contained in an equivalence class represented by region node R, we introduce pointers from v to R, and vice versa. When a query `cdequiv`(w) is made, the query engine follows these pointers to find all the cities in the equivalence class of w. Note that a query of the form 'Are cities v and w in the same equivalence class?' can be answered in constant time using this representation.

A straightforward computation and pairwise comparison of the $|V|$ `conds` set takes time $O(|V|^2|A|)$ [FOW87]. In Section 5.3, we obtain a $O(|V|+|A|)$ time algorithm by showing that a `conds` set is uniquely identified by two functions: $|A_v|$, its size, and $\mathrm{Lo}(A_v)$, a descendant of v as defined below; both these functions are computable in linear time, as shown in Section 5.2. Thus, the two functions act as *finger prints* of their sets, and a `cdequiv` set simply collects nodes with the same finger prints.

5.1 Finger prints of `cdequiv` sets

For notational convenience, we augment the tree with a distinguished node, denoted by ∞, which is considered to be a descendant of all other nodes, and by a node, denoted by $-\infty$, which is considered to be an ancestor of all other nodes.

Definition 11 *If R is a set of chariot routes,* $\mathrm{Lo}(R)$ *is defined to be the least common ancestor (lca) of the bottom nodes of routes in R. By convention,* $\mathrm{Lo}(R)$ *is ∞ if R is empty.*

This definition is more general than we need because the sets of routes we deal with always have at least one node in common. For this special case, Lo can be defined more intuitively as follows.

Lemma 1 *Let R be a non-empty set of chariot routes, and let N be the set of nodes that belong to every route in R. If N is non-empty, the nodes in N are totally ordered by the ancestor relation, and* $\mathrm{Lo}(R)$ *is the lowest node in N.*

Proof:

Since the nodes in N are contained in every route $r \in R$, and the nodes in r are totally ordered the ancestor relation, it follows that the nodes in N are ordered by this relation. Let l be the lowest node in N.

Since l is contained in each route $r \in R$, l is an ancestor of the bottom node of r. By definition of Lo, this means that $\mathrm{Lo}(R)$ is a descendant of l.

For every route $r = [b, t) \in R$, b is a descendant of $\mathrm{Lo}(R)$ (definition of Lo), and t is a proper ancestor of l (definition of l) and therefore of $\mathrm{Lo}(R)$ (since $\mathrm{Lo}(R)$ is a descendant of l). Therefore, $\mathrm{Lo}(R) \in N$, which means that l is a descendant of $\mathrm{Lo}(R)$ (definition of l).

Therefore, l and $\mathrm{Lo}(R)$ are identical. □

For example, in Figure 1, $Lo(A_f)$ is c, and $Lo(A_g)$ is a. Note that for a given node v, $\mathrm{Lo}(A_v)$ is always a descendant of v, since v is contained in every route in A_v. We show next that $|A_v|$ and $\mathrm{Lo}(A_v)$ uniquely identify A_v.

Theorem 3 *Let p and q be two nodes in the tree. Sets A_p and A_q are equal if and only if $|A_p| = |A_q|$, and* $\mathrm{Lo}(A_p) = \mathrm{Lo}(A_q)$.

Proof: ($\rightarrow$) If A_p and A_q are equal, clearly so are their finger prints.
($\leftarrow$) If A_p and A_q are different, then so are p and q and one of them, say q, is not a descendant of the other. Then, one of the following cases must hold:

1. $A_p \subset A_q$. Then, clearly $|A_p| < |A_q|$.
2. $A_p - A_q \neq \emptyset$. Then, we have $\mathrm{Lo}(A_p) \neq \mathrm{Lo}(A_q)$ as shown by the following subcase analysis.
   (a) Node p is not a descendant of node q. Then, since $\mathrm{Lo}(A_p)$ and $\mathrm{Lo}(A_q)$ are descendants of p and q, respectively, it must be that $\mathrm{Lo}(A_p) \neq \mathrm{Lo}(A_q)$.
   (b) Node p is a descendant of node q. Let $\rho = [b, t)$ be a path in A_q but not in A_p. By definition, $\mathrm{Lo}(A_q)$ lies on path ρ while, by assumption, p does not. Hence, $\mathrm{Lo}(A_q)$ can not be a descendant of p and it is therefore different from $\mathrm{Lo}(A_p)$.

In conclusion, whenever A_p and A_q are different, their finger prints are also different in at least one component. □

5.2 Computing Finger Prints Efficiently

Figure 7 shows how $|A_v|$ can be computed for each node v in a single bottom-up walk of the tree, in $O(|A| + |V|)$ time. In this subsection we give an algorithm to compute $\mathrm{Lo}(A_v)$ for each node v.

Consider first the simpler problem of computing $\mathrm{Lo}(A_q)$ just for a given node q. It is natural to look for a recursive definition of $\mathrm{Lo}(A_q)$ in terms of local values computed at q, and some values propagated up from its children. If v is a descendant of q, let $A_q{\downarrow}v$ (read as 'A_q restricted to v') be the subset of routes in A_q whose bottom nodes are descendants of v. For example, in Figure 9, $A_e{\downarrow}g$ is the set of chariot routes $\{1\}$. Our algorithm will perform a bottom-up walk of the descendants of q, propagating the value $\mathrm{Lo}(A_q{\downarrow}v)$ up from each node v; notice that the value computed when retreating out of q is $\mathrm{Lo}(A_q{\downarrow}q)$ which is nothing but $\mathrm{Lo}(A_q)$. Before describing this algorithm, we introduce some ancillary values defined at each node, which can be computed during the bottom-up pass. In the formulae below, min denotes the least common ancestor of a set of nodes totally ordered by the ancestor relation, taken to be ∞ for the empty set.

Definition 12 *The following quantities are defined for each node v.*

- $h_v = \min\{t \mid [v,t) \in A\}$,
- $H_v = \min\{t \mid [b,t) \in A_v\}$,
- $(v_1, v_2, ...)$: *the children of v ordered so that H_{v_i} is an ancestor of $H_{v_{i+1}}$,*
- $t_v = \min\{h_v, H_{v_2}, v\}$.

Informally, h_v is the highest top node of any chariot route in the set of chariot routes whose bottom node is v; for example, in Figure 9, h_e is c. H_v is the highest top node of any chariot route in the set of chariot routes containing v; in Figure 9, H_e is a. For node e, v_2 is f, and H_{v_2} is b.

To understand the significance of t_v, note that a node on the tree path $[v, t_v)$ cannot be in the same `cdequiv` class as nodes that are strictly below v because it is either contained in a chariot route whose bottom node is v (this is the case if $t_v = h_v$) or it is contained in two routes that come together at v (this is the case if $t_v = H_{v_2}$). For example, the value of t_e is b because all nodes on the tree path $[e, b)$ are contained in both Route 1 and Route 3 which come together at e, so these nodes cannot be the same `cdequiv` class as a node that is a strict descendant of e.

It is straight-forward to compute the ancillary values in a bottom-up walk. We observe first that h_v is easily computed for all nodes in linear time by first initializing each h_v to ∞, and then scanning every chariot route $[b,t)$, updating the value of h_b with $\min\{h_b, t\}$, .

Lemma 2 *If v is a* leaf, *we have $H_v = h_v$; $t_v = h_v$. If v is not a leaf node, we have:*

- $H_1 = H_{v_1}$ *and* $H_2 = H_{v_2}$ *are the* min *and second* min *in the sequence of (possibly repeated) values $(H_{c_1}, H_{c_2}, ...)$, where $c_1, c_2, ...$ are the children of v;*
- $H_v = \min(H_1, h_v)$;
- $t_v = \min(H_2, h_v, v)$.

Given these ancillary values, the following recursive formula computes the value of $Lo(A_q{\downarrow}v)$ for all nodes v that are descendants of a query node q. By convention, for a leaf node v, $Lo(A_q{\downarrow}v_1)$ is always ∞.

Lemma 3 *For a fixed q, the following formula computes $Lo(A_q{\downarrow}v)$ for each descendant v of q.*

IF $q \in [v, t_v)$
THEN $Lo(A_q{\downarrow}v) = v$;
ELSE $Lo(A_q{\downarrow}v) = Lo(A_q{\downarrow}v_1)$.

Proof: Suppose $q \in [v, t_v)$. There are two cases.

1. $t_v = h_v$. Then, by definition of h_v, $[v, t_v)$ is a chariot route. Since $q \in [v, t_v)$, $[v, t_v) \in A_q{\downarrow}v$, so $Lo(A_q{\downarrow}v) = v$.
2. $t_v = H_{v_2}$. Then, by definition of H_v, there are chariot routes $[u_1, H_{v_1})$ and $[u_2, H_{v_2})$ where v_1 and v_2 are children of v, and u_1 and u_2 are descendants of v_1 and v_2, respectively. These two routes both contain q and first meet at v. Therefore, $Lo(A_q{\downarrow}v) = v$.

Suppose q is not contained in $[v, t_v)$. This means that every route in $A_q{\downarrow}v$, if any, originates in the subtree rooted at v_1, which means that $A_q{\downarrow}v \subseteq A_q{\downarrow}v_1$. Moreover, since all routes in $A_q{\downarrow}v_1$ contain q, they must contain v. Therefore, $A_q{\downarrow}v = A_q{\downarrow}v_1$. which implies that $Lo(A_q{\downarrow}v) = Lo(A_q{\downarrow}v_1)$. □

The value of $Lo(A_q{\downarrow}q)$ is the desired value $Lo(A_q)$. Therefore, a walk over the array of chariot routes (to compute h_v) and then a single bottom-up walk of the descendants of q suffices to compute the value of $Lo(A_q)$.

We now extend this scheme to compute $Lo(A_q)$ for all nodes q. For a given q, we propagated the single value $Lo(A_q{\downarrow}v)$ up from each node v which is a descendant of q. To extend this scheme, we propagate a *sequence* of values out of each node v, where the sequence encodes the values of $Lo(A_q{\downarrow}v)$ *for every node q that is an ancestor of v*. For example, in Figure 9, the ancestors of e in bottom-up order are $< e, d, c, b, a >$, and the corresponding sequence of $Lo(A_q{\downarrow}e)$ values is $< e, e, e, i, \infty >$. Since it is too expensive to have duplicate values in the sequence, we propagate instead a sequence of pairs of the form $[x, y)$ where x is a Lo value and y is the ancestor of v where this value is no longer relevant. In our example, out of node e, we propagate the sequence of pairs $S_e =< [e, b), [i, a), [\infty, -\infty) >$. Read from left to right, this states that the $Lo(A_q{\downarrow}e)$ value is e for any ancestor q of e upto (but not including) node b, is i from there to node a, and is ∞ after that. With this interpretation, it is clear that for any node q, the value of $Lo(A_q)$ is the first element of the first pair in the sequence S_q.

If v is a node, the sequence S_v can be expressed in terms of the sequence S_{v_1} where v_1 is the child of v described in Definition 12. By convention, S_{v_1} for a leaf node v is $< [\infty, -\infty) >$.

Lemma 4 *For any node v, the sequence S_v can be computed from S_{v_1} as follows.*

1. *From S_{v_1}, delete every entry of the form $[x, y)$ where y is a descendant of t_v.*
2. *If $[v, t_v)$ is not empty, make $[v, t_v)$ the first element of the remaining sequence.*

The resulting sequence is S_v.

Proof: The proof of correctness follows immediately from Lemma 3, since $Lo(A_q){\downarrow}v \neq Lo(A_q){\downarrow}v_1$ iff $q \in [v, t_v)$. □

For any node q, the value of $Lo(A_q)$ is the first element of the first pair in the sequence S_q. Therefore, a single bottom-up pass is adequate to compute $Lo(A_q)$ for all nodes q. A key observation for efficiency is that by definition, the sequence of second elements of pairs in any S_q are totally ordered by ancestorship. For example, in S_e, the sequence of second elements is $< b, a, -\infty >$. This means that the deletion of entries of the form $[x, y)$ in Step 1 of Lemma 4 can start from the first pair in the sequence and stop as soon as we come across a y that is not a descendant of t_v. In other words, sequences can be manipulated like stacks. This is the key to obtaining an efficient implementation.

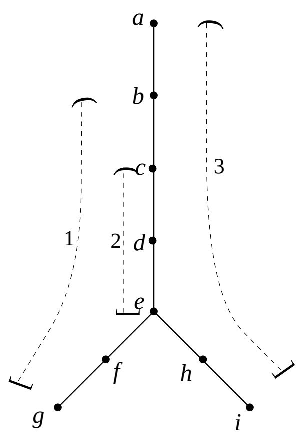


Figure 9: Computing Lo Efficiently

5.3 A fast algorithm for `cdequiv`

Figure 10 shows the pseudocode for a fast algorithm that exploits the pair of finger prints $\{\mathrm{Lo}, Size\}$ discussed above to identify nodes in the same `cdequiv` class. We use integers to identify `cdequiv` classes. Procedure **new-region-node** () returns a new integer each time it is called. This can be implemented using a static variable initialized to zero that is incremented and returned each time the procedure is called.

We assume each node structure has the following fields:

- *S[v]* — stack of node pairs
- *H[v]* — top node closest to root of any route originating from a descendant of node v.
- *RecentSize[v]* — $|A_w|$ where w is node for which $Lo(A_w) = v$ most recently in bottom-up walk. This field is initialized to 0.
- *RecentRegionNode[v]* — region node of node w where w is node for which $Lo(A_w) = v$ most recently in bottom-up walk.

To determine the complexity of this algorithm, we note that the work required to compute t_v at a node v is some constant amount plus two terms, one proportional to the number of children of v, and the other proportional to the number of routes whose bottom node is v. Summing over all nodes, we get a term that is $O(|V| + |A|)$. Next, we estimate the work required for pushing and popping pairs. At each node v, we pop a number of pairs, test one pair that is not popped, and then optionally push one pair. Since each pair is pushed once and popped once, the total cost of pushing and popping is proportional to the number of pairs, which is $O(|V|)$. Finally, the cost of testing a pair that is not popped is charged to the cost of visiting the node. Therefore, the complexity of the overall algorithm is $O(|V| + |A|)$; for the special case of the control dependence problem, this expression is $O(|E|)$. The bottom-up traversal for computing the `cdequiv` relation can be folded into the `conds` preprocessing of Section 4, but we have shown it separately for simplicity.

```
Procedure CdEquivPreprocessing(T)
{
    /* T is the postdominator tree */
    for each node v in bottom-up order do
        /* compute t[v] */ ;
        /*min returns infinity (i.e. N + 1) if the set is empty.*/
        h := min {t | [v, t) is a chariot route } ;
        h_below := min { H[c] | c is a child of v};
        H[v] := min {h, h_below} ;
        v_1 := any child c of v having H[c] = h_below ;
        H_{v_2} := min { H[c] | c is a child of v other than v_1} ;
        t_v := min { h, H_{v_2}, v };

        S[v] := S[v_1];
        From S[v], pop all pairs [x, y) where y is descendant of t_v;
        if [v, t_v) is not empty then push [v, t_v) onto S[v] endif ;
        Lo := first element of first pair in S[v];

        /* Determine class for node v */
        RecentSize[v] := 0;
        if Lo = ∞ then Class[v] := EmptyClass ;
        else /*a[v] is |A_v| */
            if RecentSize[Lo] ≠ a[v]) then
                RecentSize[Lo] := a[v];
                RecentRegionNode[Lo] := new-region-node () ;
            endif
        Connect v and RecentRegionNode[Lo] ;
        endif
    endfor

}
```

Figure 10: Algorithm for identifying `cdequiv` classes

5.4 Related Work

There is a large body of previous work on algorithms for computing the `cdequiv` relation. Ferrante et al gave the first algorithm for this problem [FOW87] — they computed the `conds` set of every node explicitly, and used hashing to determine set equality. The complexity of this algorithm is $O(|V|^2|A|)$. This algorithm was later improved by Cytron, Ferrante and Sarkar who described a quadratic time algorithm for determing the `cdequiv` relation [CFS90]. A linear time algorithm for the `cdequiv` problem for *reducible* control flow graphs was given 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]. Newburg *et al* use encodings of paths from `START` to each node to determine the `cdequiv` relation [NNS94]. They do not describe the complexity of their algorithm in terms of the size of the CFG, but it is likely to be $O(|V|^2|A|)$, if not worse.

The first optimal solution to the general `cdequiv` problem was given by Johnson, Pearson and Pingali who designed an algorithm which required $O(|E|)$ preprocessing time and space, and which enumerated `cdequiv` sets in proportional time [JPP94]. This algorithm required neither dominator nor postdominator information, since it used a depth-first tree obtained from the undirected version of the control-flow graph, in which the analogs of chariot routes were back edges in the depth-first tree. The algorithm was 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 in here. The finger prints of sets of chariot routes used in this paper are essentially identical to those in our earlier algorithm. Other researchers have started to study cycle equivalence; for example, Rauch has developed a dynamic algorithm for computing cycle equivalence incrementally when the control flow graph is modified [Rau94]. A more detailed discussion of cycle equivalence can be found in Richard Johnson's PhD thesis [Joh94].

Finally, we note that the ancillary quantities, H_v and t_v, which were introduced in Definition 12, are interesting in their own right. The tree of a Roman Chariots problem can be viewed as the DFS tree of an undirected graph, in which each chariot route $[b, t)$ representing a back edge connecting nodes b and t. For some node v, suppose that H_{v_1} is not ∞. Then H_{v_1} is the highest ancestor of v that is connected to some proper descendant of v by a back edge. To see the significance of H_v, note that $H_v = \min\{h_v, H_{v_1}\}$. If H_v is not ∞, then H_v is the highest ancestor of v reachable by a path that contains only descendants of v (other than H_v itself). It is also easy to see that if H_v is not ∞, then there are two paths from v to H_v that have no vertices in common other than v and H_v. One path is the tree path $[v, H_v]$. If $H_v = h_v$, then the other path is the back edge from v to h_v. If $H_v = H_{v_1}$, there is a proper descendant d of v such that there is a back edge from d to H_v; in that case, the other path is the tree path $[v, d]$ concatenated with the back edge from d to H_v. The existence of two node disjoint paths between v and H_v means that these nodes are in the same *biconnected component* of the undirected graph; in fact, the computation of H_v is the key step in the Hopcroft and Tarjan algorithm for computing biconnected components [AHU74] since it determines articulation points in the undirected graph. It can be shown that the computation of t_v arises similarly in the computation of *triconnected* components.

6 Implementation and Experiments

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. $R[v]$: node. Region node (`cdequiv`) associated with 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 8(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 6, 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.

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 11(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 11(a,b)).

Consider the graph for $\alpha = 1/32$ in Figure 11(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 11(b) which shows storage requirements for $\alpha > 1$. Finally, Figure 11(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 for $n = 100$, and it clearly lies above the graph of storage actually used.

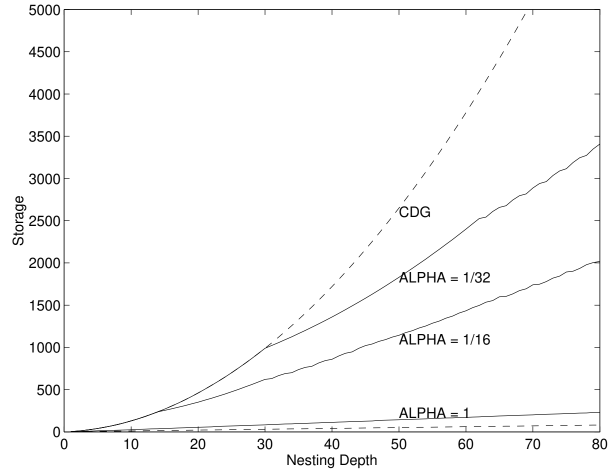
Figure 11(d) shows that for a fixed problem size, worst case query time decreases as α decreases. Because actual query time is too small to measure accurately, we measured instead the number of routes examined during querying (say r), and the number of nodes in the subzone of the query node, other than 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 11(d).

Finally, Figures 11(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.

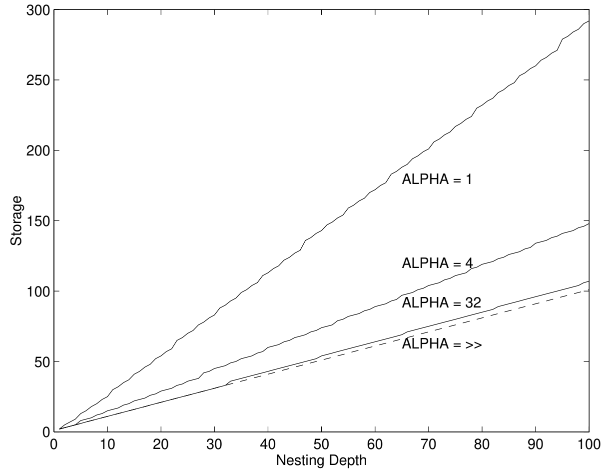
Real programs, such as the SPEC benchmarks, are less challenging than the model problem. Figure 12(a) shows a plot of storage vs. program size for all the procedures in the SPEC benchmarks. The x-axis is the number of basic blocks in a procedure, and is a measure of procedure size. The y-axis shows the total number of routes stored at all nodes of an $\mathcal{APT}$ data structure for that procedure. For each procedure, the $\mathcal{APT}$ data structure was constructed with three different values of α — (i) a very small value of α (full caching), (ii) $\alpha = 1$, and (iii) a very large value of α. From Figure 12(a), we can show that storage requirements can be reduced by a factor of 3 by using a large α. Figure 12(b) shows the total storage requirements for all the procedures in each of the SPEC benchmarks.

For a fixed problem size, the use of a very small value of α is similar to building the CDG; therefore, the data points for small α in Figure 12(a) can be viewed as the storage requirements for the CDG. Note that unlike in the model problem, storage requirements of procedures in the SPEC benchmarks grow *linearly* with problem size (this observation has been made before by other researchers [CFR$^+$91]). This can be explained as follows. It is easy to verify that if the height of the postdominator tree does not grow with problem size, the size of the CDG will grow only *linearly* with problem size. As is seen in Figure 12, the height of the postdominator tree for procedures in SPEC is quite small, and it is more or less independent of the size of the procedure (only the procedure *iniset.f* in *doduc* has a postdominator tree height of more than 75). This reflects the fact that deeply nested loops and long sequential chains of code are rare in real programs.

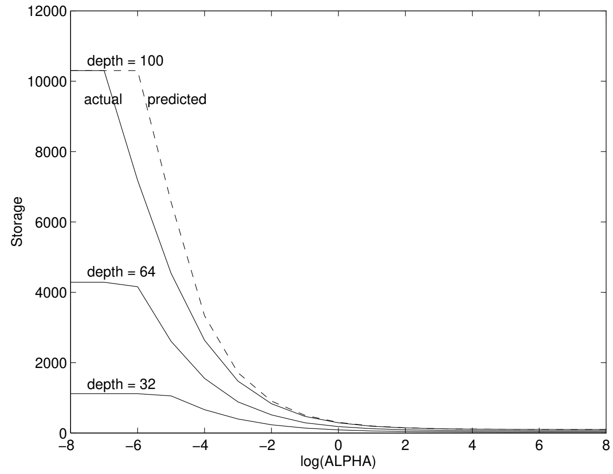
Query time was not significantly affected when α was set to 1; for larger values of α, query time for a few nodes was affected, but on the whole, the effect was small. Finally, for every procedure in the SPEC benchmarks, preprocessing time to construct $\mathcal{APT}$ is a small fraction of the time to build the postdominator tree.



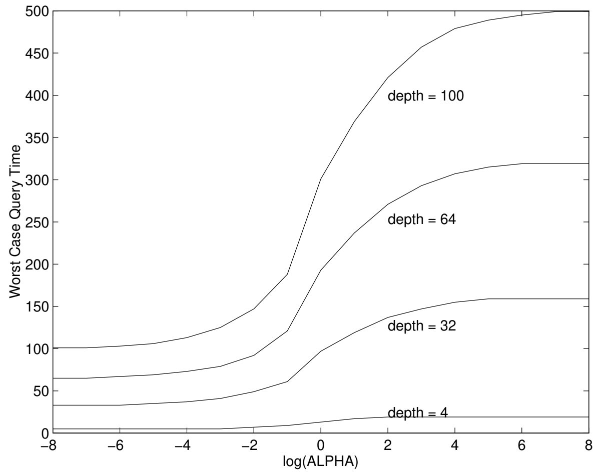
(a)Storage vs. Nesting Level ($\alpha \leq 1$)



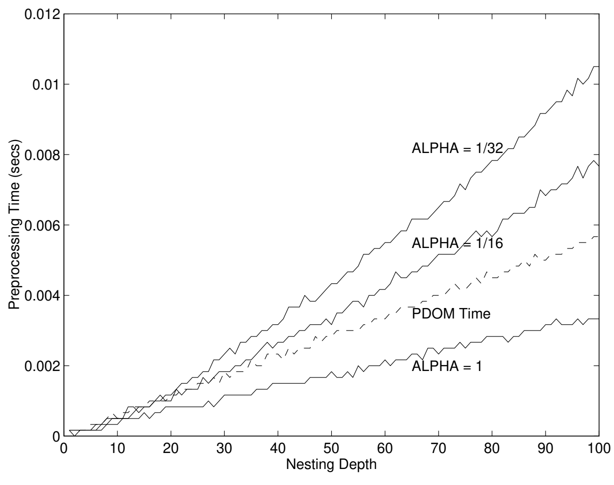
(b) Storage vs. Nesting Level ($\alpha \geq 1$)



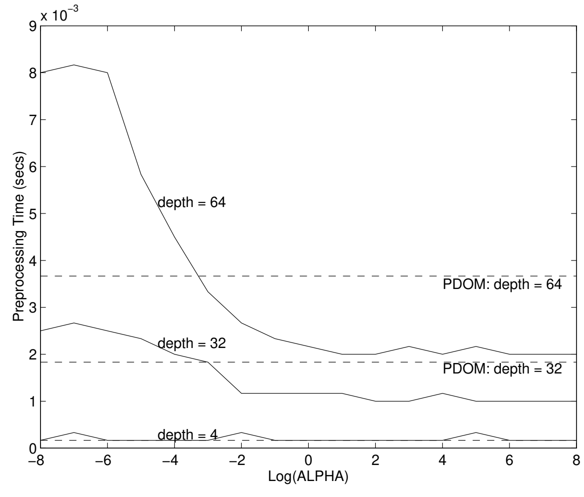
(c) Storage vs. α



(d) Query Time vs. α



(e) Preprocessing Time vs. Nesting Level



(f) Preprocessing Time vs. α

Figure 11: Experimental Results for Repeat-Until Loop Nests

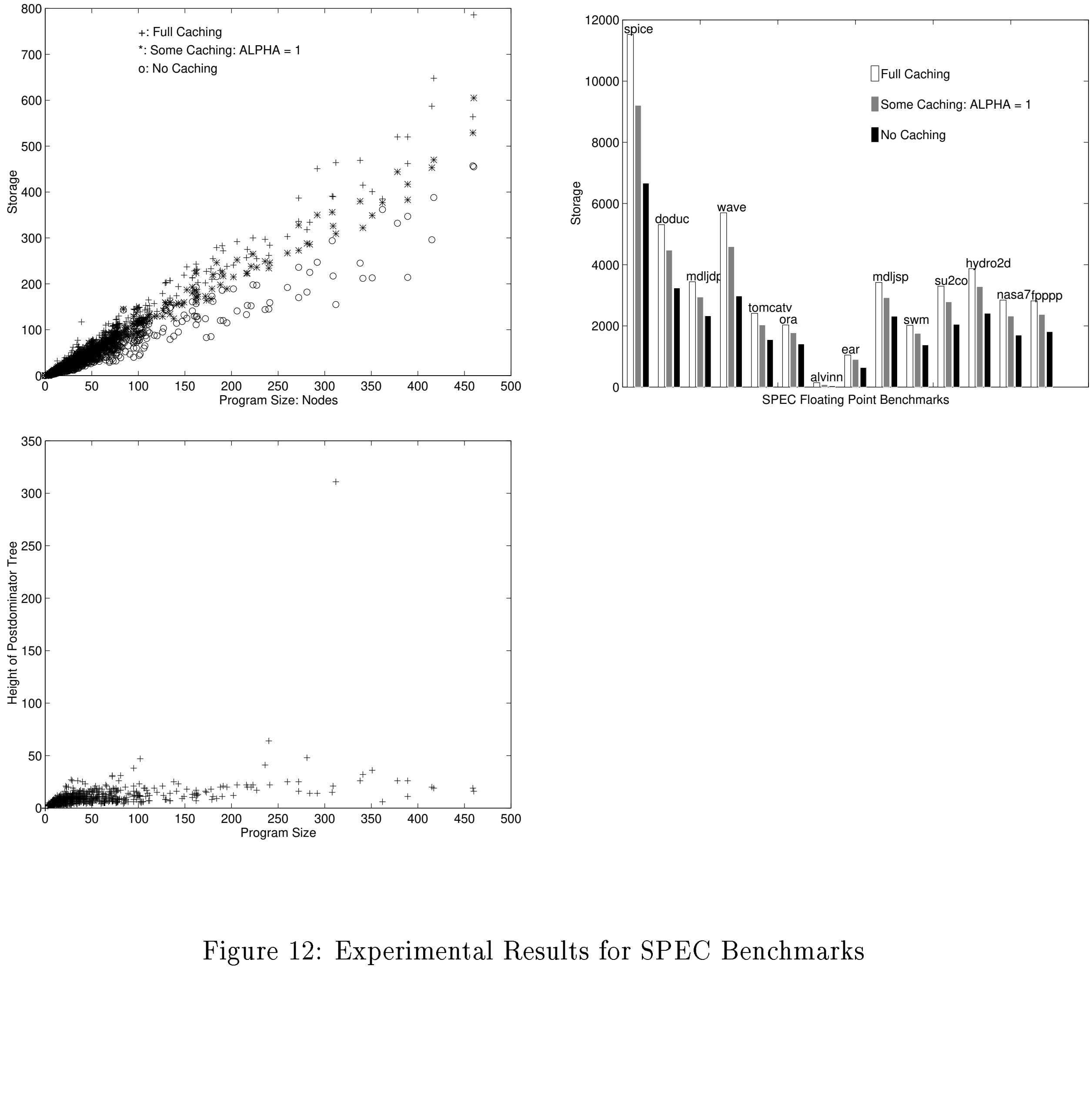


Figure 12: Experimental Results for SPEC Benchmarks

7 Conclusions

The $\mathcal{APT}$ data structure can be used to build the SSA form of a program in $O(|E|)$ time per variable by exploiting the connection between dominance frontiers and `conds` sets [PB95]. This improves the quadratic time complexity of the commonly used algorithm of Cytron *et al* [CFR+91], and it has the same complexity as a recent algorithm due to Sreedhar and Gao [SG95]. The advantage of our algorithm over the Sreedhar and Gao algorithm is that $\mathcal{APT}$ permits us to compute the dominance frontier of a node optimally, whereas the Sreedhar and Gao algorithm requires $O(|E|)$ time for this problem. On the SPEC benchmarks, this advantage results in our algorithm running 5 times faster than the Sreedhar and Gao algorithm. Furthermore, our algorithm subsumes both the Cytron *et al* algorithm and the Sreedhar and Gao algorithm — if we build $\mathcal{APT}$ with a small value of α, our algorithm reduces to the algorithm of Cytron *etal*, while a large value of α results in the algorithm of Sreedhar and Gao!

There are many alternatives to the zone construction algorithm given in this paper. 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 $\texttt{conds}(v)$, we can answer instead the query $\texttt{conds}(w)$ 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 storage requirements.

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. Of course, nothing prevents us from using run time caching together with $\mathcal{APT}$, if this is useful in some application.

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: Richard Johnson and David Pearson participated in early research that led to this paper. In particular, the `cdequiv` algorithm described here uses ideas which were developed jointly with Johnson and Pearson. Paul Chew, Mayan Moudgill, Michael Wolfe, Eric Stoltz, Barbara Ryder, V. Sreedhar, Richard Schooler and Ruth Pingali gave us invaluable feedback.

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.

[AHU74] A. V. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[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.

[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.

[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.

[NNS94] C.J. Newburn, Derek Noonburg, and John Shen. A pdg-based tool and its use in analyzing program control dependences. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, August 1994.

[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.

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

[Rau94] Monika Rauch. Fully dynamic cycle-equivalence in graphs. In *Foundations of Computer Science*, 1994.

[SAF90] Barbara Simons, David Alpern, and Jeanne Ferrante. A foundation for sequentializing parallel code. In *Conference Record of SPAA '90: ACM Symposium on Parallel Algorithms and Architecture*, July 1990.

[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.

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