Optimal Control Dependence Computation and the Roman Chariots Problem

KESHAV PINGALI
Cornell University
and
GIANFRANCO BILARDI
Università di Padova, Italy, and University of Illinois, Chicago

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 article, 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 $\texttt{cd}(e)$, which is the set of statements control dependent on control-flow edge e, of the set $\texttt{conds}(w)$, which is the set of edges on which statement w is dependent, and of the set $\texttt{cdequiv}(w)$, 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 filtering search.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers and optimization*; I.1.2 [**Algebraic Manipulation**]: Algorithms—*analysis of algorithms*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Compilers, control dependence, program optimization, program transformation

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

Keshav Pingali was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9503199, and ONR grant N00014-93-1-0103. Gianfranco Bilardi 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.

Authors' addresses: K. Pingali, Department of Computer Science, Cornell University, Ithaca, NY 14853; email: pingali@cs.cornell.edu; G. Bilardi, DEI, Università di Padova, Padova, Italy; Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60607; email: bilardi@art.dei.unipd.it.

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 1.1. A **control flow graph** $G = (V, E)$ is a directed graph in which nodes represent statements, and an edge $u \to v$ 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 [Ferrante et al. 1987].

Definition 1.2. 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 Lengauer and Tarjan [1979]; $\alpha(|E|)$ denotes the inverse Ackermann function which grows extremely slowly with $|E|$ so that the algorithm can be considered linear for all practical purposes. This algorithm is relatively easy to code. A rather more complicated algorithm due to Harel [1985] computes the postdominator tree optimally in $O(|E|)$ time. Control dependence can be defined formally as follows:

Definition 1.3. A node w is said to be **control dependent** on edge $(u \to v)$ 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 $u \to v$, it will eventually reach node w; however, control may reach `END` from u without passing through w. Thus, u is a "decision-point" that influences the execution of w.

Definition 1.4. 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.

The notion of control dependence is due to Ferrante et al. [1987]. Cytron et al. [1990] studied many of the properties of this relation. These authors define control dependence as a relation between nodes (in the context of Definition 1.3, they view w as being control dependent on node u rather than on the edge $(u \to v)$). The definition given in this article is easier to work with, but the difference is merely one of presentation, not substance.

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 [Cytron et al. 1990]:

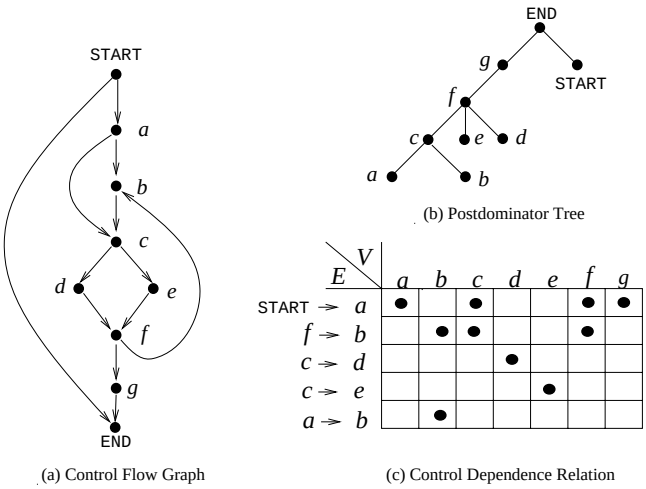


Fig. 1. A program and its control dependence relation.

Definition 1.5. 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**:

—$\texttt{cd}(e) = \{w \in V | (e, w) \in C\}$,
—$\texttt{conds}(w) = \{e \in E | (e, w) \in C\}$, 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 [Bernstein and Rodeh 1991; Fisher 1981; Newburn et al. 1994] and are used in merging program versions [Horowitz et al. 1987]. They are also useful in automatic parallelization [Allen et al. 1988; Ferrante et al. 1987; Simons et al. 1990]. 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 [Gupta and Soffa 1987]. 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 [Johnson 1994; Johnson et al. 1994] and to perform dataflow analysis in parallel [Gupta et al. 1990; Johnson et al. 1994].

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 that $\texttt{cdequiv}(a) = \texttt{cdequiv}(g) = \{a, g\}$; all the other nodes are in `cdequiv` sets by themselves.

In this article, 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,

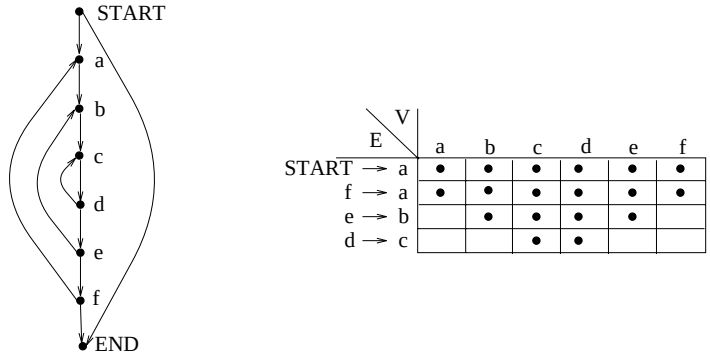


Fig. 2. A program with three nested repeat-until loops.

—*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 can be $\Omega(|V||E|)$, 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 [Harel 1985; Lengauer and Tarjan 1979] 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 (1) postdominance is a transitive relation and (2) 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) [Ferrante et al. 1987], 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 the control dependence relation [[Ball 1993]; Cytron et al. 1990; Ferrante et al. 1987; Johnson and Pingali 1993; Sreedhar et al. 1994]. The lack of success led Cytron et al. to conjecture that any data structure

that provided proportional time access to control dependence sets must use space that grows quadratically with program size.

In this article, we describe a data structure called the *Augmented Postdominator Tree* ($\mathcal{APT}$) which requires $O(|E|)$ space, is built in $O(|E|)$ time,[1] and provides 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, corresponding to a "more explicit" representation of the control dependence relation. Interestingly, the full control dependence graph can be viewed as one extreme of this range of data structures, obtained when $\alpha \leq 1/|E|$.

The rest of the article 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 [Michie 1968]; we also show that our approach can be viewed as an example of Chazelle's *filtering search* [Chazelle 1986].

2. THE ROMAN CHARIOTS PROBLEM

We show that the computation of control dependence sets (Definition 1.5) 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 [Ferrante et al. 1987]. First, we introduce some convenient notation.

Definition 2.1. 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.)

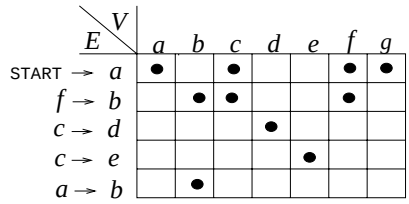
For example, in the postdominator tree of Figure 1(b), $[d, a] = \{d, f, c, a\}$, while $[d, a) = \{d, f, c\}$. This notation is similar to the standard one for open and closed intervals of the line. The following key theorem, due to Ferrante et al. [1987], shows how edges of the control flow graph are constrained with respect to the postdominator tree and provides a simple characterization of `cd` sets.

THEOREM 2.2. *If $(u \to v)$ is an edge of the control flow graph, then*

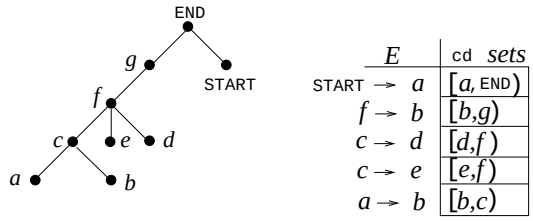
(1) $parent(u)$ is an ancestor of v in the postdominator tree and
(2) $\texttt{cd}(u \to v) = [v, parent(u))$.

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

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



(a) Control dependence relation



(b) Postdominator tree and cd sets

Fig. 3. Compact representation of control dependence.

(1) If $parent(u)$ does not postdominate v, we can find a path $v \rightarrow ... \rightarrow$ END which does not contain $parent(u)$. Prefixing this path with the edge $u \rightarrow v$, 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 $\texttt{cd}(u \rightarrow v) \subseteq [v, parent(u))$. Let w be an element of $\texttt{cd}(u \rightarrow v)$. 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 $u \rightarrow v$.

This gives a concise characterization of cd sets. □

Figure 3 shows the nonempty cd sets for the program of Figure 1(a). 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.[2] The postdominator tree and the array of cd sets, together, can be viewed as

[2]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

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, they 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 article.

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 nonempty. 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.[3] Nodes represent cities, and edges represent roads. 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) `cd`(ρ): the *cities* on route ρ.
(2) `conds`(w): the *routes* that serve city w.
(3) `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 2.3. 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:

—`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).

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 article.

[3]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 (Cicero, 56 B.C., Pro L. Cornelio Balbo Oratio 39), just as in a tree rooted at Rome.

```
Procedure ConstructRomanChariots(G:CFG):Tree, RouteArray;
{
1:    % G is the control flow graph
2:    T := construct-postdominator-tree(G);
3:    A := [ ]; %Initialize to empty array
4:    i := 0;
5:    for each node p in T in top-down order do
6:    for each child u of p do
7:       for each edge (u → v) in G do
8:          if v is not p
9:             then %append a cd set to end of A
10:               i := i + 1;
11:               A[i] := [v, p);
12:               Note the correspondence between
13:               edge u → v and index i;
14:         endif
15:      od
16:   od
17:   return T, A;
}
```

Fig. 4. Constructing a Roman Chariots problem.

—`conds`(w): Execute the Roman Chariots query `conds`(w), and translate each integer (between 1 and 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 2.2 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 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* ($\mathcal{APT}$). The rest of the article establishes the following result.

THEOREM 2.4. *There is a data structure $\mathcal{APT}$ for the Roman Chariots problem $(T =< V, F, ROME >, A)$ that 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:*

—`cd`(e)*: time $O(|$*`cd`$(e)|)$*, independent of α.*

—`conds`(w)*: time $O((1 + \alpha)|$*`conds`$(w)|)$.

—`cdequiv`(w)*: time $O(|$*`cdequiv`$(w)|)$*, independent of α.*

For the special case when T is the postdominator tree of a control flow graph, we have $|A| \leq |E|$ and $|V| \leq |E| + 1$, leading to the following result.

COROLLARY 2.5. *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 `cd`(i), where i is between 1 and m (the size of A), let $[v, w)$ be the ith 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. [1987].

4. $\mathcal{APT}$: `CONDS` QUERIES

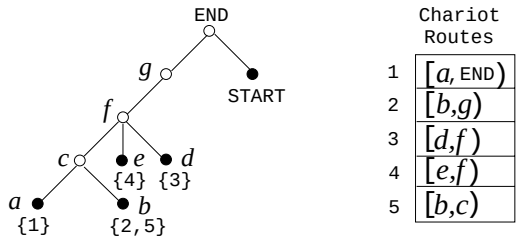
One way to answer `conds` queries is to examine all routes in array A and report every route whose bottom node is a descendant of the query node and whose top node is a proper ancestor of the query node. This algorithm is too slow.

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.

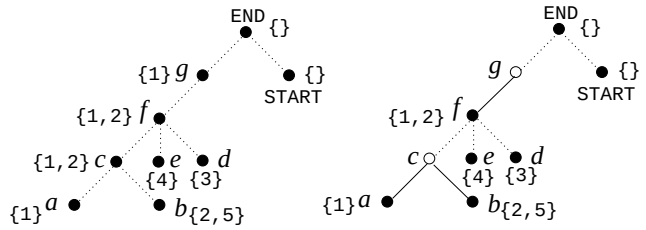
It is possible to compromise between these two extremes. Suppose we partition the nodes in V into two disjoint sets called *boundary* nodes and *interior* nodes. Although this partition can be made arbitrarily, it is simpler to make all leaf nodes boundary nodes; for now, nonleaf nodes can be classified 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 4.1. 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 nonleaf nodes



(a) No caching



(b) Full caching (c) Some caching: $\alpha = 1$

Fig. 5. Zone structure.

are interior nodes, while Figure 5(b) shows the other extreme in which 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 4.1, 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 4.2. A node w is said to be in the *subzone* associated with a node q, referred to as Z_q, if (1) w is a descendant of q and (2) the path [q,w) does not

```
Procedure CondsQuery(QueryNode);
{
1:    % APT data structure is global variable;
2:    % Query outputs list of routes numbers
3:    CondsVisit(QueryNode, QueryNode);
}
Procedure CondsVisit(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:               CondsVisit(QueryNode,C)
13:            od ;
14:   endif ;
15:
}
```

Fig. 6. Query procedure for `conds`.

contain any boundary nodes.

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

In Figure 5(c), there are six zones induced by the following sets of nodes: $\{a, b, c\}$, $\{d\}$, $\{e\}$, $\{f, g\}$, {START}, and {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 endpoint, 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 then establish ancestorship by comparing *dfs* numbers. Since *dfs* numbers are already assigned by postdominator tree construction algorithms [Harel 1985; Lengauer and Tarjan 1979], 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:

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

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 positive 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 Eqs. (1) and (2), we see that

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

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 a small number close to 0 (strictly speaking, a number less than $1/|A|$ where $|A|$ is the number of chariot routes), 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 make zones as large as possible without violating Inequality (2). More precisely, a leaf node is always a boundary node. For a nonleaf 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.[4] Formalizing this intuitive description, we obtain a definition for subzone construction.

[4]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.

Definition 4.1.1. **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

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

From Definition 4.1.1, 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 Z_u subzones involved in the resulting double summation are disjoint. Hence, $\sum_{v\in X}|A_v| \le |V|/\alpha$, which, used in Relation (4), yields

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

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 2.4.

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 $\alpha \ge |V|$, 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.

```
Procedure CondsPreprocessing(T:tree,A:RouteArray,α:real);
{
1:  % b[v]/t[v]: number of routes with bottom/top node v
2:  for each node v in T do
3:     b[v] := t[v] := 0;
4:  od
5:  for each route [x, y) in A do
6:     Increment b[x];
7:     Increment t[y];
8:  od
9:  %Determine boundary nodes.
10: for each node v in T in bottom-up order do
11:    %Compute output size when v is queried.
12:    a[v] := b[v] - t[v] + Σ_{u∈children(v)} a[u];
13:    z[v] := 1 + Σ_{u∈children(v)} z[u]; %Tentative zone size.
14:    if (v is a leaf) or (z[v] > α * a[v] + 1)
15:       then % Begin a new zone
16:          Bndry?[v] := true;
17:          z[v] := 1;
18:       else %Put v into same zone as its children
19:          Bndry?[v] := false;
20:    endif
21: od
22: % Chain each node to the first boundary node that is an ancestor.

23: for each node v in T in top-down order do
24:    if v is root of postdominator tree
25:       then NxtBndry[v] := - ∞;
26:       else if Bndry?[parent(v)]
27:          then NxtBndry[v] := parent(v);
28:          else NxtBndry[v] := NxtBndry[parent(v)];
29:          endif
30:    endif
31: od
32: % Add each route in A to relevant L[v]
33: for i := 1 to |A| do
34:    let A[i] be [b, t);
35:    w := b;
36:    while t is proper ancestor of w do
37:       append i to end of list L[w];
38:       w := NxtBndry[w];
39:    od
40: od
}
```

Fig. 7. Constructing the $\mathcal{APT}$ structure.

—The array of routes, A, in which routes are sorted by top endpoint. For control dependence problems, this array is built by Procedure **ConstructRomanChariots** shown in Figure 4.

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

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

(1) *For each node v, compute the number of routes whose bottom node is v and the number of routes whose top node is v.* Let $b[v]$ be the number of routes in A whose bottom endpoint is v, and let $t[v]$ be the number of routes whose top endpoint is 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 4.1.1 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.

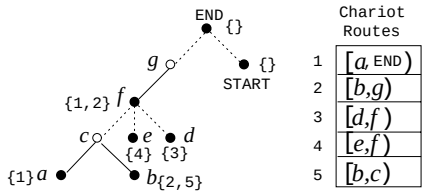
(5) *Construct list $L[v]$ for each node v.* By Definition 4.1, a given route $[b, t)$ appears in list $L[v]$ for $v \in W$, where W contains b as well as all boundary nodes contained by $[b, t)$. Specifically let $W = \{w_0 = b, w_1, ..., w_k\}$, where $w_i = NxtBndry[w_{i-1}]$ for $i = 1, 2, ..., k$ and where w_k is the proper descendant of t such that t is a descendant of $NxtBndry[w_k]$. $L[v]$ lists are formed by scanning the routes in A where 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 is $T = O(|A| + (1 + 1/\alpha)|V|)$, as claimed.

Figure 7 shows the pseudocode for building the search structure. All the preprocessing, including construction of the route array A, can be done in two top-down and one bottom-up walks of the postdominator tree, followed by one traversal of the route array. Figure 8 illustrates the $\mathcal{APT}$ data structure (with $\alpha = 1$) for the control flow graph of Figure 1.

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



(a) Caching: $\alpha = 1$

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

Fig. 8. The $\mathcal{APT}$ structure and its parameters.

are served by the same set of routes. In this section, we describe a preprocessing algorithm that produces a list representation of the equivalence classes where each list cell points to the next cell as well as to the header of its list. A query `cdequiv`(w) is answered by simply traversing the list containing w, starting from the header. A query of the form "Are cities v and w in the same equivalence class?" can be answered in constant time by checking whether w and v have the same header.

A straightforward computation and pairwise comparison of the $|V|$ `conds` set takes time $O(|V|^2|A|)$ [Ferrante et al. 1987]. Exploiting structure leads to faster algorithms for some problems, but we are not aware of any structure in `cdequiv` sets. In particular, nodes in a `cdequiv` equivalence class are not necessarily adjacent either in the control flow graph or in the postdominator tree (consider nodes a and g in Figure 1). This suggests that we cannot afford to compute and compare `conds` sets explicitly if we want a fast algorithm. 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 $Lo(A_v)$, 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 *fingerprints* of their sets, and a `cdequiv` set simply collects nodes with the same fingerprints.[5]

The fingerprints we use are easily understood if we restrict attention to the special

[5]An analogy from physics is the identification of elements from their spectra rather than directly from their atomic structure.

case when the tree is a chain. A single bottom-up walk is sufficient to compute the *size* of the `conds` set of each node in the chain. Treating this integer as a fingerprint, we can form equivalence classes by placing all nodes with `conds` sets of the same size into one class. This gives a coarser equivalence relation than the `cdequiv` relation because the `conds` sets of nodes p and q are not necessarily equal just because these sets are of the same size. Therefore, we need another fingerprint. Let p be an ancestor of q, and suppose that $A_p \neq A_q$. If $|A_p| = |A_q|$, there must be some route $\rho = [b, t) \in A_p$ which is not contained in A_q. Therefore, b must be a proper ancestor of q. This suggests the second fingerprint. For each node v, compute the node $Lo(A_v)$ which is defined to be the lowest node contained in *all* the routes in A_v, and place nodes in the same equivalence class only if this fingerprint is the same for both nodes. In our example, $Lo(A_p)$ will be an ancestor of b, while $Lo(A_q)$ will be a proper descendant of b, so p and q will have different fingerprints.

It is easily seen that the two fingerprints *set size* and *Lo* together determine the `cdequiv` equivalence relation completely. The rest of the section shows how to compute these fingerprints quickly in general trees.

5.1 Fingerprints 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 5.1.1. If R is a set of chariot routes, $Lo(R)$ is defined to be the least common ancestor (lca) of the bottom nodes of routes in R. By convention, $Lo(R)$ 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 5.1.2. *Let R be a nonempty set of chariot routes, and let N be the set of nodes that belong to every route in R. If N is nonempty, the nodes in N are totally ordered by the ancestor relation, and $Lo(R)$ 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 by 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 $Lo(R)$ is a descendant of l.

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

Therefore l and $Lo(R)$ are identical. □

For example, in Figure 5(b), $Lo(A_f)$ is c, and $Lo(A_g)$ is a. Note that, for a given node v, $Lo(A_v)$ is always a descendant of v, since v is contained in every route in A_v. We show next that $|A_v|$ and $Lo(A_v)$ uniquely identify A_v.

THEOREM 5.1.3. *Let p and q be two nodes in the tree. Sets A_p and A_q are equal if and only if both of the following are true: $|A_p| = |A_q|$ and $Lo(A_p) = Lo(A_q)$.*

PROOF.

($\rightarrow$). If A_p and A_q are equal, clearly so are their fingerprints.

($\leftarrow$). If A_p and A_q are different, then so are p and q. There are two cases to consider.

(1) Nodes p and q are not related by the ancestor relation. Since $Lo(A_p)$ and $Lo(A_q)$ are descendants of p and q respectively, it follows that $Lo(A_p) \neq Lo(A_q)$.

(2) Node q is a descendant of node p. Suppose that $|A_p| = |A_q|$. Then, there must be some route $\rho = [b, t)$ that contains p but not q. $Lo(A_p)$ must be an ancestor of b, and q must be a proper descendant of b. Since $Lo(A_q)$ is a descendant of b, it follows that $Lo(A_p) \neq Lo(A_q)$. Therefore, either $|A_p| \neq |A_q|$ or $Lo(A_p) \neq Lo(A_q)$.

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

5.2 Computing Fingerprints 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 $Lo(A_v)$ for each node v.

Consider first the simpler problem of computing $Lo(A_q)$ just for a given node q. It is natural to look for a recursive definition of $Lo(A_q)$ 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 $Lo(A_q{\downarrow}v)$ up from each node v; notice that the value computed when retreating out of q is $Lo(A_q{\downarrow}q)$, which is nothing but $Lo(A_q)$. 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 5.2.1. 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 $H_{v_{i+1}}$, and
—$t_v = \min\{h_v, H_{v_2}, v\}$.

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, for two reasons: 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

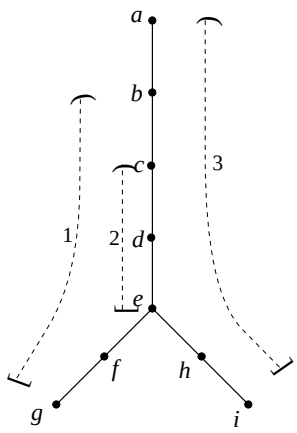


Fig. 9. Computing *Lo* efficiently.

(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 straightforward 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 5.2.2. *If v is a* leaf, *we have $H_v = h_v$; $t_v = h_v$. If v is not a leaf node, we have the following:*

—$H_1 = H_{v_1}$ *and* $H_2 = H_{v_2}$ *are the* min *and second* min *in the sequence of* (*possibly repeated*) *values* $(H_{v_1}, H_{v_2}, ...)$, *where* $v_1, v_2, ...$ *are the children of* v;

—$H_v = \min(H_1, h_v)$; *and*

—$t_v = \min(H_2, h_v, v)$.

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 q, $Lo(A_q{\downarrow}v_1)$ is always ∞.

LEMMA 5.2.3. *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 where 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 suffice 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 that 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 up to (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 5.2.1. By convention, S_{v_1} for a leaf node v is $< [\infty, -\infty) >$.

LEMMA 5.2.4. *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 5.2.3, 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 5.2.4 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.

5.3 A Fast Algorithm for `cdequiv`

Figure 10 shows the pseudocode for a fast algorithm that exploits the pair of fingerprints $\{Lo, Size\}$ (discussed above) to identify nodes in the same `cdequiv` class. Each class is represented as a linked list of nodes; each node points to the next one in the list as well as to the header of the list. The ability to link a node in constant time rests on the following observation. Let $u_1, u_2, \ldots, u_k$ be a set of nodes, ordered from descendant to ancestor, all with the same Lo, i.e., $Lo(u_1) = Lo(u_2) = \ldots = Lo(u_k)$. Then, it can be easily seen that the corresponding $Size$ values form a nonincreasing sequence, i.e., $|A_{u_1}| \geq |A_{u_2}| \geq \ldots \geq |A_{u_k}|$. The sequence $u_1, u_2, \ldots, u_k$ is then naturally broken into segments representing `cdequiv` classes, the breaking points being those where a strict inequality occurs.

We assume each node structure has the following six fields, the first four being auxiliary to the computation of the last two, which encode the `cdequiv` classes and are part of the $\mathcal{APT}$:

—*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.

—*RecentNode*[*v*]: node w for which $Lo(A_w) = v$ most recently in bottom-up walk. This field is initialized to v.

—*CdEqNext*[*v*]: Successor of v in the list of nodes representing `cdequiv`(v).

—*CdEqHeader*[*v*]: Header of the list of nodes representing `cdequiv`(v).

Since every node is control dependent on at least one edge, the Roman Chariots problem derived from a control dependence problem has the property that every city is contained in at least one chariot route. For a general Roman Chariots problem, this is not necessarily the case, and it is necessary to put all nodes that are not contained in any route into one equivalence class. It is natural to use node ∞ as the header for this class. This processing can be omitted for control dependence problems.

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)
{
1:  /* T is the postdominator tree */
2:  /* Processing of node ∞ not needed for control dependence
    problems */
3:  RecentSize[∞] := 0;
4:  RecentNode[∞] := ∞;
5:  CdEqNext[∞] := null;
6:  CdEqHeader[∞] := ∞;
7:  for each node v in bottom-up order do
8:     /* compute t[v] */ ;
9:     /*min returns infinity (i.e. N + 1) if the set is empty.*/
10:    h := min  {t | [v, t) is a chariot route } ;
11:    h_below := min  { H[c] | c is a child of v};
12:    H[v] := min  {h, h_below} ;
13:    v1 := any child c of v having H[c] = h_below ;
14:    Hv2 := min  { H[c] | c is a child of v other than v1} ;
15:    tv := min  { h, Hv2, v };
16:    S[v] := S[v1];
17:    From S[v], pop all pairs [x, y) where y is descendant of tv;
18:    if [v, tv) is not empty then push [v, tv) onto S[v] endif ;
19:    Lo := first element of first pair in S[v];
20:    /* Determine class for node v */
21:    RecentSize[v] := 0;
22:    RecentNode[v] := v;
23:    CdEqNext[v] := null;
24:    /*a[v] is |Av| */
25:    if RecentSize[Lo] = a[v] then
26:       /* add to current cdequivclass */
27:       CdEqNext[RecentNode[Lo]] := v;
28:       CdEqHeader[v] := CdEqHeader[RecentNode[Lo]]
29:    else /* start a new cdequivclass */
30:       RecentSize[Lo] := a[v];
31:       CdEqHeader[v] := v
32:    endif
33:    RecentNode[Lo] := v;
34: od
}
```

Fig. 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. [1987] gave the first algorithm for this problem: 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 et al. [1990] who described a quadratic-time algorithm for determining the `cdequiv` relation. A linear-time algorithm for the `cdequiv` problem for *reducible* control flow graphs was given by Ball [1993] who needed both dominator and postdominator information in his solution; subsequently, Podgurski [1993]

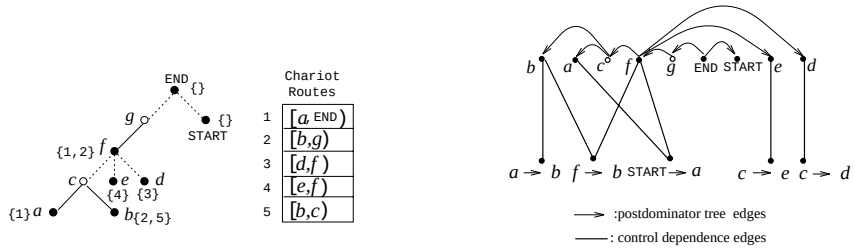
gave a linear-time algorithm for *forward* control dependence equivalence, which is a special case of general control dependence equivalence. Newburg et al. [1994] use encodings of paths from `START` to each node to determine the `cdequiv` relation. They do not describe the complexity of their algorithm 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 et al. [1994] who designed an algorithm that required $O(|E|)$ preprocessing time and space and that enumerated `cdequiv` sets in proportional time. 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 nontrivial 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 fingerprints of sets of chariot routes used in this article are essentially identical to those in Johnson et al. [1994]. Other researchers are studying properties of cycle equivalence; for example, Rauch [1994] has developed a dynamic algorithm for computing cycle equivalence incrementally when the control flow graph is modified. A more detailed discussion of cycle equivalence can be found in Johnson [1994].

Finally we note that the ancillary quantities, H_v and t_v, which were introduced in Definition 5.2.1, 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 $[b, t)$ represents 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 [Aho et al. 1974], 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:



(a) $\mathcal{APT}$: $\alpha = 1$ (b) Actual Implementation

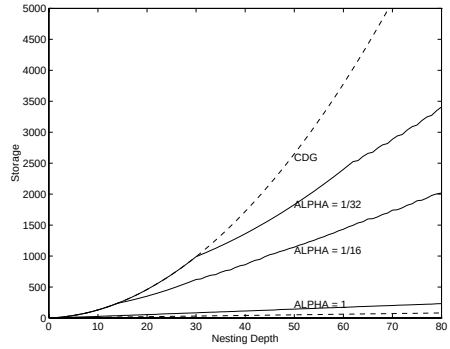
Fig. 11. Implementation of $\mathcal{APT}$.

(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) $CdEqNext[v]$: Successor of v in the list of nodes representing `cdequiv`(v).
(7) $CdEqHeader[v]$: Header of the list of nodes representing `cdequiv`(v).

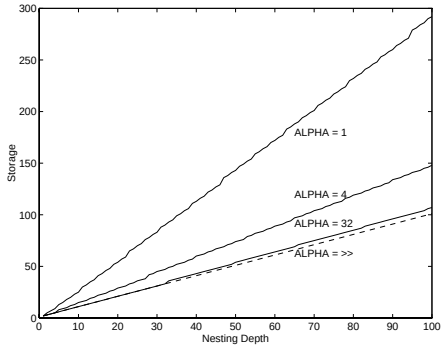
Two aspects of our $\mathcal{APT}$ implementation for the control dependence problem are worth mentioning. Instead of storing chariot routes (`cd` sets) at nodes, we store (references to) the corresponding CFG edges. For example, in Figure 11(a), chariot routes `1` and `2` are stored at node f. In the actual implementation, shown conceptually in Figure 11(b), we store references to the corresponding CFG edges, `START` $\rightarrow a$ and $f \rightarrow b$. This is convenient because it enables the output of `conds` queries to be produced directly without translation from integers to CFG edges, thereby eliminating a data structure that would be needed for this translation. Since a CFG edge $u \rightarrow v$ can be converted to a chariot route $[v, parent(u))$ in constant time, this does not affect the asymptotic complexity of any of the algorithms in the article. Finally, in procedure **CondsQuery** of Figure 6, it is worth inlining the call to procedure **CondsVisit** 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 12(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 as expected grows quadratically with problem size. For a fixed problem size, the storage needed for $\mathcal{APT}$ 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 12(a) and (b)).

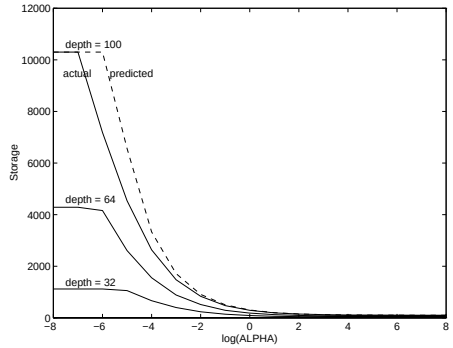
Consider the graph for $\alpha = 1/32$ in Figure 12(a). For small problem sizes (between 1 and 31), storage requirements look exactly like those of the CDG. For



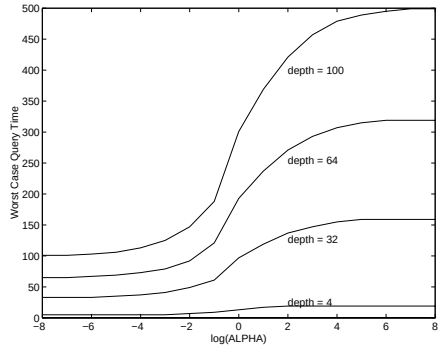
(a)Storage vs. Nesting Level ($\alpha \leq 1$)



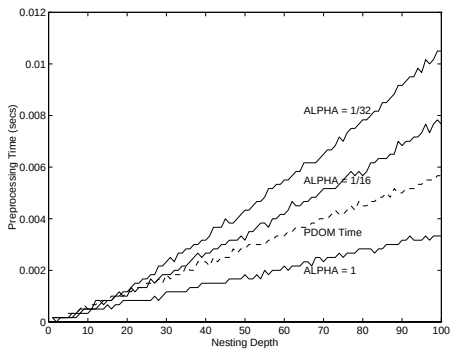
(b) Storage vs. Nesting Level ($\alpha \geq 1$)



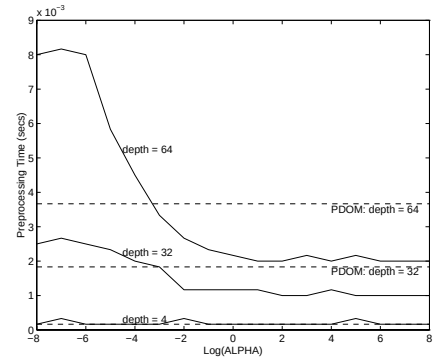
(c) Storage vs. α



(d) Query Time vs. α



(e) Preprocessing Time vs. Nesting Level



(f) Preprocessing Time vs. α

Fig. 12. Experimental results for repeat-until loop nests.

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 Eq.(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

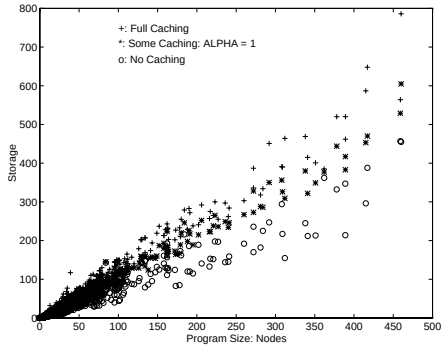
adaptive nature of the $\mathcal{APT}$ data structure. Intuitively, $1/\alpha$ is a measure of the "budget" for space — if the problem size is small compared with the budget, the algorithm performs full caching. As problem size increases, full caching becomes more and more expensive, until at some point (1) zones with more than one node start to appear and (2) the graph for $\mathcal{APT}$ peels away from the graph for the CDG. A similar analytical interpretation is possible for Figure 12(b), which shows storage requirements for $\alpha > 1$. Finally, Figure 12(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 12(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 12(d).

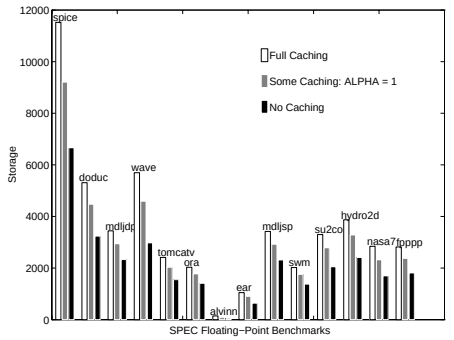
Finally, Figures 12(e) and (f) show how preprocessing time varies with problem size and with α. These times were measured on a SUN-4. Note that for $\alpha > 1/8$, preprocessing time is less than the time to build the postdominator tree; even for very small values of α, the time to build the $\mathcal{APT}$ 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 13(a) shows a plot of storage versus 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: (1) a very small value of α (full caching), (2) $\alpha = 1$, and (3) a very large value of α. From Figure 13(a), we can show that storage requirements can be reduced by a factor of 3 by using a large α. Figure 13(b) shows the total storage requirements for all the procedures in each of the SPEC benchmarks.

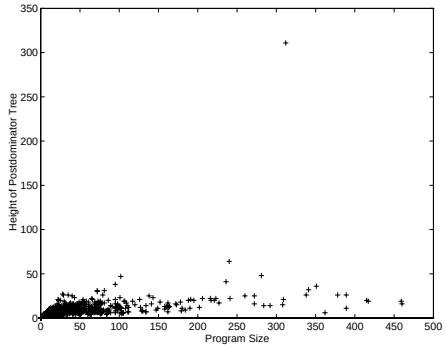
For a fixed problem size, the use of a very small value of α is similar to building the CDG; therefore the data points for small α in Figure 13(a) can be viewed as the storage requirements for the CDG. Note that, unlike in the model problem, storage requirements of procedures in the SPEC benchmarks grow *linearly* with problem size (this observation has been made before by other researchers [Cytron et al. 1991]). 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 13(c), the height of the postdominator tree for procedures in SPEC is quite small, and it is more or less



(a) Storage vs. Program Size



(b) Storage for SPEC Floating-Point Benchmarks



(c) Height of Postdominator tree vs. Program Size

Fig. 13. Experimental results for SPEC benchmarks.

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.

In general, the choice $\alpha = 1$ appears quite reasonable for a implementation of $\mathcal{APT}$.

7. CONCLUSIONS

In recent work [Bilardi and Pingali 1996], we have shown how the ideas in $\mathcal{APT}$ can be applied successfully to other interesting variants of the control dependence problem such as *weak control dependence*, which was introduced by Podgurski and Clarke [1990] for proving total correctness of programs. In particular, we provide an $\mathcal{APT}$-like data structure, constructed in $O(|E|)$ preprocessing space and time, for answering weak control dependence queries in optimal time. This improves the $O(|V|^3)$ time required by the Podgurski and Clarke algorithm.

We have also used the ideas in $\mathcal{APT}$ to build the SSA form of a program in

$O(|E|)$ time per variable by exploiting the connection between dominance frontiers and `conds` sets [Pingali and Bilardi 1995]. This algorithm improves the quadratic-time complexity of the commonly used algorithm of Cytron et al. [1991], and it has the same complexity as a recent algorithm due to Sreedhar and Gao [1995]. 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 (with α set to 1) 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 et al., while a large value of α produces the algorithm of Sreedhar and Gao.

There are many alternatives to the zone construction algorithm given in this article. 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 [Johnson et al. 1994]. 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 in 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 [Michie 1968; Segre and Scharstein 1993], 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 [Cytron et al. 1990]. Factoring identifies nodes that have control dependences in common and creates representations that 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 (1) that it can be viewed as a factored representation, since a route is cached just once per zone, and (2) that entry for

the route 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 that 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 article can be viewed as an example of *filtering search* [Chazelle 1986], 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." Filtering 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.

ACKNOWLEDGEMENTS

Richard Johnson and David Pearson participated in early research that led to this article. In particular the `cdequiv` algorithm described here uses ideas that were developed jointly with Johnson and Pearson. Paul Chew, Guong-Rong Gao, Dexter Kozen, Giuseppe Italiano, Mayan Moudgill, Ruth Pingali, Geppino Pucci, Barbara Ryder, Richard Schooler, V. Sreedhar, Eric Stoltz, Eva Tardos, and Michael Wolfe gave us invaluable feedback. We thank the referees for their constructive comments.

REFERENCES

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

Allen, F., Burke, M., Cytron, R., Ferrante, J., Hsieh, W., and Sarkar, V. 1988. A framework for determining useful parallelism. In *Proceedings of the 1988 International Conference on Supercomputing.* IEEE, New York, 207–215.

Ball, T. 1993. What's in a region? or computing control dependence regions in near-linear time for reducible control flow. *ACM Lett. Program. Lang. Syst. 2,* 1–4 (Mar.–Dec.), 1–16.

Bernstein, D. and Rodeh, M. 1991. Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation.* ACM, New York, 241–255.

Bilardi, G. and Pingali, K. 1996. A framework for generalized control dependence. In *Proceed-*

ings of the SIGPLAN '96 Conference on Programming Language Design and Implementation. ACM, New York, 291–300.

Chazelle, B. 1986. Filtering search: A new approach to query answering. *SIAM J. Comput. 15*, 703–724.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13,* 4 (Oct.), 451–490.

Cytron, R., Ferrante, J., and Sarkar, V. 1990. Compact representations for control dependence. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*. ACM, New York, 337–351.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. 1987. The program dependency graph and its uses in optimization . *ACM Trans. Program. Lang. Syst. 9,* 3 (June), 319–349.

Fisher, J. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. 7,* 3, 478–490.

Gupta, R., Pollock, L., and Soffa, M. L. 1990. Parallelizing data flow analysis. In *Proceedings of the Workshop on Parallel Compilation*. Queen's University, Kingston, Ontario.

Gupta, R. and Soffa, M. L. 1987. Region scheduling. In *2nd International Conference on Supercomputing*. IEEE, New York, 141–148.

Harel, D. 1985. A linear time algorithm for finding dominators in flowgraphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*. ACM, New York, 185–194.

Horowitz, S., Prins, J., and Reps, T. 1987. Integrating non-interfering versions of programs. In *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 133–145.

Johnson, R. 1994. Efficient program analysis using dependence flow graphs. Ph.D. thesis, Cornell Univ., Ithaca, N.Y.

Johnson, R. and Pingali, K. 1993. Dependence-based program analysis. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM, New York, 78–89.

Johnson, R., Pearson, D., and Pingali, K. 1994. The program structure tree: Computing control regions in linear time. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, New York, 171–185.

Lengauer, T. and Tarjan, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1,* 1 (July), 121–141.

Michie, D. 1968. Memo functions and machine learning. *Nature 218*, 19–22.

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

Pingali, K. and Bilardi, G. 1995. $\mathcal{APT}$: A data structure for optimal control dependence computation. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*. ACM, New York, 32–46.

Segre, A. and Scharstein, D. 1993. Bounded-overhead caching for definite-clause theorem proving. *J. Autom. Reason. 11*, 83–113.

Simons, B., Alpern, D., and Ferrante, J. 1990. A foundation for sequentializing parallel code. In *SPAA '90: ACM Symposium on Parallel Algorithms and Architecture*. ACM, New York.

Sreedhar, V. C., Gao, G. R., and Lee, Y. 1994. DJ-graphs and their applications to flowgraph analyses. Tech. Rep. ACAPS Memo 70, McGill Univ., Montreal, Canada. May.

Received July 1996; accepted October 1996