The Program Structure Tree: Computing Control Regions in Linear Time

Richard Johnson
rjohnson@cs.cornell.edu

David Pearson
pearson@cs.cornell.edu

Keshav Pingali
pingali@cs.cornell.edu

Department of Computer Science
Cornell University, Ithaca, NY 14853

Abstract

In this paper, we describe the program structure tree (PST), a hierarchical representation of program structure based on single entry single exit (SESE) regions of the control flow graph. We give a linear-time algorithm for finding SESE regions and for building the PST of arbitrary control flow graphs (including irreducible ones). Next, we establish a connection between SESE regions and control dependence equivalence classes, and show how to use the algorithm to find control regions in linear time. Finally, we discuss some applications of the PST. Many control flow algorithms, such as construction of Static Single Assignment form, can be speeded up by applying the algorithms in a divide-and-conquer style to each SESE region on its own. The PST is also used to speed up data flow analysis by exploiting 'sparsity'. Experimental results from the Perfect Club and SPEC89 benchmarks confirm that the PST approach finds and exploits program structure.

1 Introduction

The contributions of this paper are the following.

In Section 2, we introduce the *program structure tree* (PST) which is a hierarchical representation of the control structure of a program. Nodes in this tree represent *single entry single exit (SESE) regions* of the program, while edges represent nesting of regions. The PST is defined for *all* control flow graphs, including irreducible graphs.

In Section 3, we give an $O(E)$ algorithm for finding SESE regions. This algorithm works by reducing the problem to that of determining a simple graph property that we call *cycle equivalence*: two edges are cycle equivalent in a strongly connected component iff for all cycles C, C contains either both edges or neither edge. We give a fast, linear-time algorithm based on depth-first search for solving the cycle equivalence problem, thereby finding SESE regions in linear time. This algorithm runs very fast in practice — for example, our empirical results show that it runs faster than Lengauer and Tarjan's algorithm for finding dominators [LT79]. We use this algorithm to build the PST for arbitrary flow graphs in $O(E)$ time. In Section 4, we give experimental results that characterize the structure of the PST in standard benchmarks such as Perfect Club, SPEC, and Linpack programs. As one would expect, the PST is usually broad and shallow — roughly 97% of all SESE regions have a nesting depth of 6 or less.

In Section 5, we apply the cycle equivalence algorithm to finding *control regions* in $O(E)$ time. Two nodes are said to be in the same control region if they have the same set of control dependences [FOW87]. Previous algorithms for this problem are either restricted to reducible flow graphs [Bal92] or have $O(EN)$ complexity [CFS90]. Control region information is useful for problems such as instruction scheduling for pipelined machines [GS87]; therefore, our linear-time algorithm for region determination is of wide interest.

The PST is a tool which can enhance the performance of many program analysis algorithms. Each SESE region is a control flow graph in its own right, so any program analysis algorithm can be applied directly to it. The partial results from each SESE region can be combined using the PST to give the result for the entire procedure. Provided that the combining is not overly expensive, this 'divide-and-conquer' style of applying analysis algorithms can be advantageous since the PST is a natural data structure for exploiting global structure (nesting), local structure (of each SESE region), and sparsity. We make these points in Section 6 by showing how the PST can be used in three problems: conversion to SSA form, data flow analysis and dominator computation. We also discuss possible applications of the PST to parallel and incremental program analysis.

[1]This research 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. David Pearson is supported by a Fannie and John Hertz Fellowship.

2 Single entry single exit regions and the program structure tree

In the literature, the term 'single entry single exit region' is not used consistently — there appear to be several related constructs 'aliased' to this term [Kas75, Val78, TV80, GPS90]. Therefore, we begin this section with a formal definition of single entry single exit regions as used in this paper . This definition is motivated in part by considerations of control dependence, as will be made precise in Section 5. We then show that single entry single exit regions can be organized into a tree called the *program structure tree* (PST).

Figure 1(a) shows a control flow graph with its single entry single exit regions marked. Note that each SESE region is enclosed by a pair of control flow graph edges called the entry and exit edges respectively. SESE regions are either nested, sequentially composed, or disjoint. When regions are sequentially composed, the exit edge of one region is also the entry edge of the following region. Figure 1(b) shows the PST of the control flow graph of Figure 1(a). The PST captures the nesting relationship of SESE regions; chains of sequentially composed SESE regions, such as regions c, d and e, are grouped in the PST.

2.1 Defining single entry single exit regions

First, we recall a few standard definitions.

Definition 1 *A* **control flow graph** G *is a graph with distinguished nodes* `start` *and* `end` *such that every node occurs on some path from* `start` *to* `end`*.* `start` *has no predecessors and* `end` *has no successors.*

Definition 2 *A node x is said to* **dominate** *node y in a directed graph if every path from* `start` *to y includes x. A node x is said to* **postdominate** *a node y if every path from y to* `end` *includes x.*

By convention, a node dominates and postdominates itself. The notions of dominance and postdominance can be extended to edges in the obvious way. Single entry single exit regions can now be defined as follows.

Definition 3 *A* **SESE region** *in a graph G is an ordered edge pair (a,b) of distinct control flow edges a and b where*

1. *a dominates b,*
2. *b postdominates a, and*
3. *every cycle containing a also contains b and vice versa.*

We refer to a as the entry edge and b as the exit edge of the SESE region. The first condition ensures that every path from `start` into the region passes through the region's entry edge, a. The second condition ensures that every path from inside the region to `end` passes through the region's exit edge, b. The first two conditions are necessary but not sufficient to characterize SESE regions: since backedges do not alter the dominance or postdominance relationships, the first two conditions alone do not prohibit backedges entering or exiting the region. The third condition encodes two constraints: every path from inside the region to a point 'above' a passes through b, and every path from a point 'below' b to a point inside the region passes through a.

For future reference, we define the notion of *cycle equivalence*.

Definition 4 *Edges a and b are said to be* **edge cycle equivalent** *iff every cycle containing a contains b, and vice versa. Similarly, two nodes are said to be* **node cycle equivalent** *iff every cycle containing one of the nodes also contains the other.*

If (a, b) is a SESE region and (b, c) is a SESE region, then (a, c) is a SESE region as well. Therefore, a graph with E edges can have $O(E^2)$ SESE regions — for example, every edge pair in a linear sequence of nodes encloses a SESE region. However, we have never found any use for complete enumeration of all SESE regions of a graph. Instead, for each edge e in the graph, we want to find the smallest SESE regions, if they exist, for which e is an entry edge or an exit edge. We will call these the *canonical* SESE regions associated with e. We express this more formally as follows.

Definition 5 *A SESE region (a, b) is* **canonical** *provided*

- *b dominates b' for any SESE region (a, b'), and*
- *a postdominates a' for any SESE region (a', b).*

In straightline code, the region between any two points is single entry single exit; we will ignore these trivial regions and focus on SESE regions in the *block-level* CFG, in which straightline code sequences have been coalesced into basic blocks. Every edge in the block level CFG is either between a control operator (switch or merge) and a basic block, or between two control operators.

2.2 The program structure tree

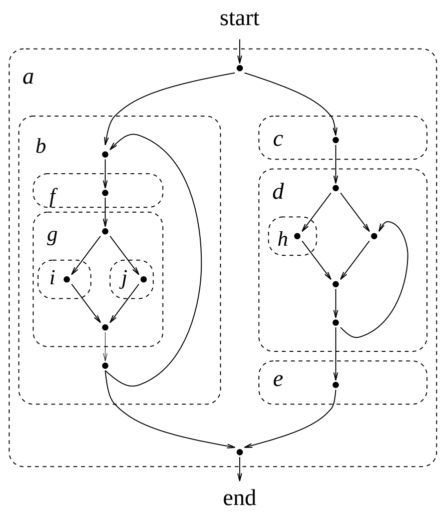
We now consider the nesting structure of canonical SESE regions and their organization into the program structure tree.

Definition 6 *A node n in a graph G is* **contained** *within the SESE region (a, b) if a dominates n and b postdominates n.*

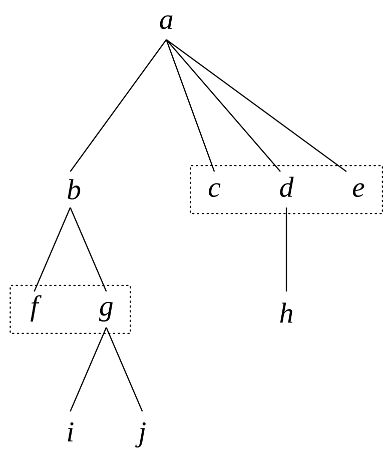
Intuitively, node n is 'between' a and b in the graph. This definition can be extended in the obvious way to containment of SESE regions. Theorem 1 describes how canonical SESE regions in a graph are related.

Theorem 1 *If R_1 and R_2 are two canonical SESE regions of a graph, one of the following statements applies.*

1. *R_1 and R_2 are node disjoint.*
2. *R_1 is contained within R_2 or vice versa.*



(a) control flow graph with SESE regions



(b) program structure tree

Figure 1: A control flow graph and its program structure tree

In other words, canonical SESE regions cannot have any partial overlap — if two regions have any nodes in common, they are either nested or in tandem.[2] This is obvious in the case of structured programs. For general control flow graphs, the required result may be proved as follows.

Proof: Suppose distinct canonical SESE regions (a, b) and (s, t) both contain a node n. Since a and s both dominate n, they are ordered by dominance. Without loss of generality, assume a dominates s. Similarly, b and t both postdominate n, so they are ordered by postdominance. If b postdominates t, then (s, t) is contained within (a, b).

Otherwise, t postdominates b. There are now three cases to consider; in each case we derive a contradiction.

1. b and s are the same edge. Note that an edge cannot both dominate and postdominate a node. Since b postdominates n and s dominates n, this case cannot happen.

2. b and s are distinct and s dominates b. Since a dominates s and s dominates b, there is a b-free path from `start` to a to s. Therefore, every path from s to `end` must contain b since otherwise we would have a b-free path from a to s to `end` which contradicts the fact that b postdominates a. Therefore, b postdominates s. Similarly, s postdominates a; otherwise there is a s-free path from `start` to a to b to `end`, which contradicts the fact that s dominates b.

   Every cycle through b passes through a and therefore contains a path from a to b; this path must contain s since a dominates s which dominates b. Therefore, every cycle through b passes through s. A similar argument shows that every cycle through s must contain t and therefore b. Therefore, (s, b) is a SESE region; since s postdominates a, it follows that (a, b) is not canonical which is a contradiction.

3. b and s are distinct and s does not dominate b. Then there is a s-free path from `start` to b. This means that s must postdominate b; otherwise, we have a path from `start` to b to `end` which passes through t (since t postdominates b) but not s, violating the assumption that s dominates t. Since b postdominates n, s also postdominates n. But s dominates n. Since an edge cannot both dominate and postdominate a node, this is a contradiction.

□

[2]Notice that this property may not be true for SESE regions that are not canonical.

In Figure 1, regions b and c are disjoint, regions a and b are nested, and regions f and g are sequentially composed.

It follows from Theorem 1 that SESE regions can be organized as a tree. Each node in this tree represents a SESE region. The parent of a region is the closest containing region, and children of a region are all the regions immediately contained within it. We call this the *program structure tree* (PST). We now show how the PST can be determined in $O(E)$ time.

3 Building the PST in linear time

The algorithm has two steps — first, find SESE regions and second, organize canonical SESE regions into the PST.

3.1 Cycle equivalence is adequate

To find SESE regions, it is convenient to reduce the three conditions for SESE regions to the single property of cycle equivalence in a related graph.

Theorem 2 *In a control flow graph G, edges a and b enclose a single entry single exit region if and only if a and b are cycle equivalent in the graph formed from G by adding an edge from* `end` *to* `start`.

Proof: [$\Rightarrow$] Suppose a and b enclose a SESE region in control flow graph G. By definition, a and b are cycle equivalent in G; we must show they are cycle equivalent in S, the strongly connected graph formed by adding edge `end` $\rightarrow$ `start` to G. Consider any cycle in S not in G. Such a cycle is formed by a path from `start` to `end` together with the backedge `end` $\rightarrow$ `start`. If this cycle contains a, then it also contains b since b postdominates a. Similarly, if this cycle contains b, then it also contains a since a dominates b. Therefore, a and b are cycle equivalent in S.
[$\Leftarrow$] Suppose a and b are cycle equivalent in S. Then a and b are cycle equivalent in G since every cycle in G is also a cycle in S. Now consider any path P from `start` to `end` containing both a and b; such a path exists since every edge occurs on some path from `start` to `end`, and since a and b are cycle equivalent in S. Without loss of generality, assume a occurs first on this path. There can be no b-free path from a to `end`, since this would yield a cycle in S containing a but not b (using the portion of P from `start` to a, the b-free path to `end`, and the backedge from `end` to `start`). Therefore, b postdominates a, and the portion of P from the last occurrence of b to `end` is a-free. There can be no a-free path from `start` to b, since this would yield a cycle in S containing b but not a. Therefore a dominates b. $\square$

3.2 From directed to undirected graphs

Further simplification is possible because of the rather surprising result that cycle equivalence in a strongly connected graph remains the same when edge directions are removed. This result allows us to find cycle equivalence classes in the undirected multigraph corresponding to a strongly connected graph. The advantage of working with undirected graphs is that algorithms based on depth-first search are simplified in undirected graphs since cross edges and forward edges are eliminated.

Theorem 3 *Let S be a strongly connected component, and let U be the undirected multigraph formed from S by removing edge directions. Edges a and b are cycle equivalent in S if and only if the corresponding undirected edges a' and b' are cycle equivalent in U.*

Proof: [$\Rightarrow$] We show that if edges a' and b' are not cycle equivalent in U, then corresponding edges a and b are not cycle equivalent in S. Without loss of generality, assume there is at least one cycle in U containing a' but not b'. Each edge on such a cycle has an associated direction in S. Adjacent edges in the cycle either have the same direction or opposing directions; if adjacent edges have opposing directions, we say there is a *direction change* at the node between these edges.

Choose C' to be a cycle in U containing a' but not b' such that this cycle has a minimum number of direction changes. If C' has no direction changes, then the corresponding edges in S form a directed cycle containing a but not b. Otherwise, C' has some minimum, non-zero number of direction changes.

Traversing C' from a' along the direction of a, let x and y be the nodes on C' where edge direction first changes and then changes back. Since S is strongly connected, there exists a directed path in S from x to y; let E' be the corresponding

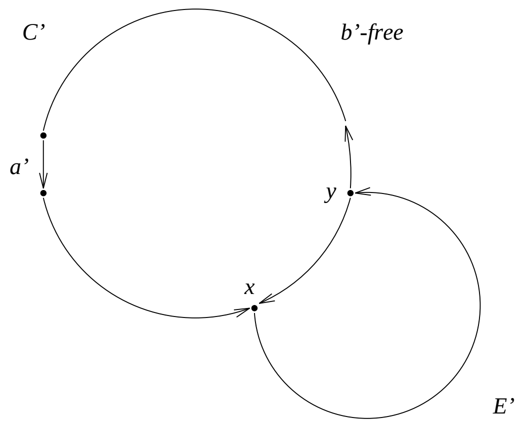


Figure 2: Undirected cycle C' and path E', with edge directions shown

undirected path in U (Figure 2). Suppose neither a' nor b' occur on E', and consider the cycle obtained by replacing the portion of C' between x and y with path E'. The resulting cycle contains a' but not b' and has fewer direction changes than C', contradicting the assumption that C' has a minimum number of direction changes.

Otherwise, a' and b' may occur (perhaps several times) on E'. If the first occurrence on E' is a', then the path from a' to x along C' together with the path along E' from x to the first occurrence of a' corresponds to a directed b-free cycle through a. Similarly, if the last occurrence of either a' or b' on E' is a', the path along E' from the last occurrence of a' to y, together with the path from y to a' along C', forms a b'-free cycle through a' having fewer direction changes than C'.

Otherwise, the first and the last occurrence of either a' or b' on E' are both b'. The path from b' to y along E', y to x along C', and then x to b' along E' corresponds to a directed a-free cycle through b.
[$\Leftarrow$] Suppose a and b are not cycle equivalent in S. Without loss of generality, there is a directed cycle in S containing a but not b. The corresponding undirected cycle in U contains a' but not b', so a' and b' are not cycle equivalent in U. $\square$

3.3 A slow algorithm for cycle equivalence

Given a strongly connected graph S, let U be the undirected multigraph formed by removing edge directions. Since U is connected, a depth-first traversal will yield a depth-first spanning tree, and the edges of U are divided into a set of tree edges and a set of backedges. Notice that any cycle in U must contain at least one backedge. We use this observation to recast the problem of cycle equivalence in terms of sets of *backedges* rather than sets of cycles.

Definition 7 *In any depth-first traversal of U, a* **bracket** *of a tree edge t is a backedge connecting a descendant of t to an ancestor of t.*

Now consider whether two edges in U are cycle equivalent. Two backedges cannot be cycle equivalent since the cycle formed from a backedge together with the tree path connecting its endpoints contains no other backedges. On the other hand, a tree edge and a backedge or two tree edges may be cycle equivalent. The following theorems establish conditions for detecting these equivalences.

Theorem 4 *A backedge b and a tree edge t are cycle equivalent if and only if b is the only bracket of t.*

Proof: [$\Rightarrow$] Suppose b and t are cycle equivalent in U. Since b together with the tree path connecting its endpoints forms a cycle, b must be a bracket of t. No other backedge can be a bracket of t, since such a backedge together with the tree path connecting its endpoints would form a cycle containing t but not b.

[$\Leftarrow$] Suppose b is the only bracket of t. Then b is the only backedge connecting a descendant of t to an ancestor of t, and since every cycle must contain a backedge, any cycle through t must contain b. Any cycle through b is comprised of b together with a b-free path connecting b's endpoints. Any such path must contain t, since every b-free path to a descendant of t must pass through t. □

The following lemma is needed to prove when two tree edges are cycle equivalent; the condition for equivalence and proof follow.

Lemma 1 *In a depth-first spanning tree of U, if tree edges s and t have any bracket in common then they are ordered by the ancestor relation in the tree.*

Proof: (by contradiction) Suppose s and t are not ordered by the ancestor relation. Then no descendant of s is a descendant of t and vice versa. Any bracket of s connects a descendant of s (say node x) to an ancestor of s; since x is not a descendant of t, this cannot be a bracket of t. □

Theorem 5 *Tree edges s and t are cycle equivalent in U if and only if they have the same set of brackets in any depth-first spanning tree of U.*

Proof: [$\Rightarrow$] We show that if two tree edges do not have the same set of brackets in a depth-first spanning tree of U, they are not cycle equivalent. Suppose edge b is a bracket of s but is not a bracket of t. By the definition of brackets, s must occur on the tree path connecting the endpoints of b, but t does not; this tree path together with b forms a cycle containing s but not t.
[$\Leftarrow$] If s and t have the same set of brackets, Lemma 1 asserts that they are ordered by the ancestor relation in the depth-first spanning tree. Without loss of generality, assume s is an ancestor of t. Any cycle through s must contain at least one backedge connecting a descendant of s to an ancestor of s. Let b be the first such backedge after s on the cycle; note that all nodes in the cycle between s and b are descendants of s. Since b is a bracket of s it is also a bracket of t, and so t is on the tree path between s and the lower[3] endpoint of b. If the cycle path from s to b does not contain t, there must be some edge (p, q) on the path which bypasses t, i.e. p is an ancestor of t and q is a descendant of t. However, both p and q are descendants of s. So (p, q) is a bracket of t but is not a bracket of s, a contradiction. The proof that every cycle containing t contains s is similar and is omitted. □

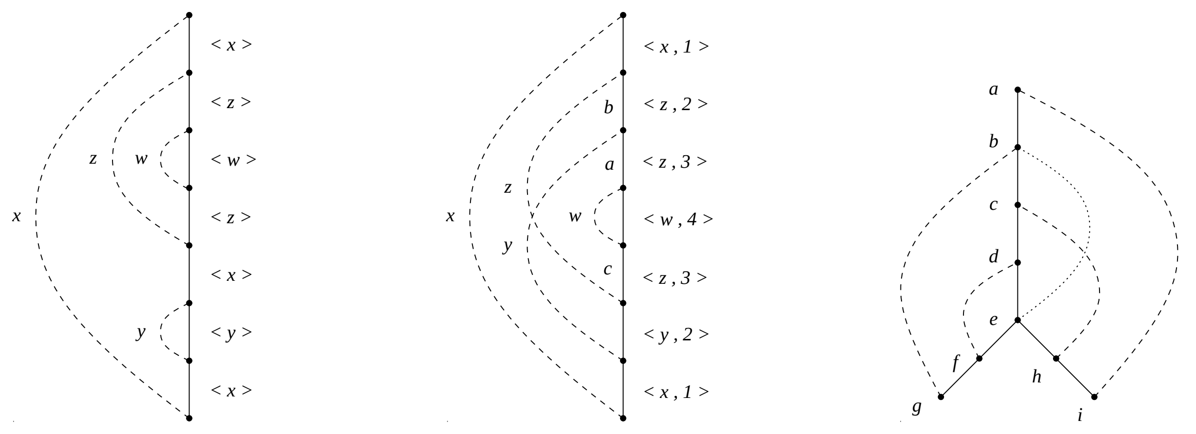
[3]Throughout this section, we use variations of *high* and *low* to refer to relative positions in the depth-first search tree. *Higher* locations are closer to root and have smaller DFS numbers.

During an undirected depth-first traversal, we can compute the set of brackets for each tree edge. When retreating out of a node, we form the union of bracket sets from the node's children, together with the set of backedges from the node to an ancestor, minus the set of backedges from a descendant to the node; the result is the bracket set for the tree edge into the current node. Intuitively, the set of brackets of a tree edge is a name for the edge's cycle equivalence class; by comparing these sets, we find cycle equivalent edges. However, building and comparing sets is expensive, so the algorithm is inefficient. In the next section, we describe a compact naming scheme for bracket sets that allows us to avoid building and comparing entire sets.

3.4 Compact names for sets of brackets

Consider the graph shown in Figure 3(a) in which the depth-first spanning tree is a simple chain and backedges correspond to 'structured' loops that are are either disjoint or nested within each other. For such graphs, it is easy to see that the set of brackets of an edge is uniquely named by the innermost bracket of that edge, so the entire bracket set at each tree edge is not needed. Instead, we can simply visit nodes in reverse depth-first order and maintain a *stack* of brackets. At each node, we delete brackets that connect a descendant to the current node, and we add any brackets connecting the node to an ancestor. Since the backedges are disjoint or properly nested, the deletions and insertions all occur at the top of the bracket stack. When retreating out of a node, the tree edge from its parent is labeled with the name of the topmost bracket in the bracket stack; after this traversal, tree edges with the same bracket label belong to the same equivalence class. In Figure 3(a), each tree edge is labeled with the topmost element of the bracket stack, and cycle equivalent edges have the same label.

Now consider the slightly more general case of linear spanning trees in which the backedges are not properly nested; an example is shown in Figure 3(b). The difficulty here is that in the reverse depth-first traversal, brackets are not deleted in stack order. Moreover, note that edges a and b do not have the same set of brackets even though the topmost element of the bracket stack of both edges is z. To allow arbitrary deletion, we implement the bracket stack with a doubly-linked list. Brackets are always added to the top of the stack, but they may be deleted from any position within the stack. In this way, the most recently added bracket (the bracket whose lower endpoint is highest in the tree) will be at the top of the stack. In addition, we will keep track of the *size* of the bracket stack. It is easy to see that the pair $< topmost\ bracket, set\ size >$ uniquely labels each equivalence class — for example, in Figure 3(b), edges a



(a) structured loops (b) unstructured loops (c) general tree node

Figure 3: Compact names for bracket sets

and b will be placed in different equivalence classes, while edges a and c are placed in the same equivalence class.

Finally, we must handle general depth-first spanning trees; an example is shown in Figure 3(c). When we encounter a node that has more than one child, the bracket sets of the children must be merged. Unfortunately, the notion of 'innermost bracket' is no longer well-defined. For example, at node e in Figure 3(c), it is not clear whether the most-recently added backedge should be edge (f, d) or edge (h, c). The resolution of this difficulty rests on the observation that only one of the subtrees below node e can contain any edges cycle equivalent to an ancestor of e. This is because an edge in a subtree of e can only have brackets originating in the same subtree; therefore, any ancestor of e having brackets from multiple subtrees of e cannot be cycle equivalent with any descendant of e. For example, edges between e and b cannot be cycle equivalent to any edge below e. However, edges between b and a can be cycle equivalent to edges between h and i.

The solution therefore is to add an additional "capping" backedge whenever we need to merge two or more bracket sets. This backedge becomes the topmost bracket in the set, and the children's bracket sets are then concatenated in arbitrary order. The new bracket originates from the node whose children are being merged, and extends up to the highest node whose brackets come from more than one of the branches. To add this new backedge requires keeping track (at each node in the tree) of the highest node reached by any backedge below this point. The destination of the new backedge from a node is the *second-highest* of the node's children's highest backedges. This could be found by examination of the bracket sets, but the highest-ending backedge is not necessarily related to the first bracket in each set (the highest-*originating*), so a full search of the bracket set would be necessary. Fortunately, we can simply compute this information independently in constant time for each node. In Figure 3(c), we would add a new backedge from e to b, as shown by the dotted edge. We must show that once this backedge is added, the pair $< topmost\ bracket, set\ size >$ identifies the equivalence class as before.

Lemma 2 *The capping backedges added by the algorithm do not alter the cycle equivalence relation for tree edges.*

Proof: By Theorem 5, two tree edges are cycle equivalent if and only if they have the same set of brackets. Consider tree edges s and t. If they have the same set of brackets after adding capping backedges, then they have the same set of brackets without adding capping backedges. We must show that they will share the same set of new brackets when capping backedges are added.

We will use the example in Figure 3(c) for illustration. Suppose edge s is bracketed by a capping backedge (e, b). The origin of that backedge, e, is a node with at least two children: the highest-reaching branch has a backedge to a point at least as high in the tree as b, and the second-highest-reaching branch has a backedge to b. Now consider where edge t (which has the same original set of brackets as s) can occur. Edge t must be within the bracket (g, b) from the second-highest-reaching subtree of e, so t must be somewhere on the tree path from g to b; t must also be within the bracket (i, a) from the highest-reaching subtree of e, so t must occur on the tree path from i to a. The intersection of these two paths is the tree path from e to b. Thus, the new bracket (e, b) is a bracket of t. □

Theorem 6 *The compact bracket set names uniquely identify bracket sets.*

Proof: We need to prove that two edges will have the same compact name if and only if they are cycle equivalent. One direction is reasonably easy: if two edges are cycle equivalent, they will receive the same compact name. By Theorem 5 two cycle equivalent edges will have the same bracket sets. By Lemma 2 the backedges added during the depth-first traversal will not affect the cycle equivalence relation. Therefore the bracket sets, as computed by the algorithm, will have the same size and the same top bracket. The cycle equivalent edges will therefore receive the same compact name.

To complete the proof, we need to establish that if two edges are not cycle equivalent, then they will not receive the same compact name. Let a and b be two edges that are not cycle equivalent. By Theorem 5 they must have different bracket sets, including (by Lemma 2) the new backedges added by the algorithm. If these sets are different size, the algorithm clearly gives them different compact names, so let us suppose the bracket sets are the same size. By Lemma 1, if a and b are not ordered by the ancestor relation, then they have no brackets in common and therefore receive different compact names. Otherwise, assume without loss of generality that a is an ancestor of b. Since the sets are the same size, but not identical, a must have a bracket (p, q) not shared by b, and b must have a bracket (r, s) not shared by a. The node p is a descendant of a — if it is also an ancestor of b then the edge (p, q) must be linked on the bracket list ahead of b's top bracket. Either (p, q) will be the top bracket, or there will be another still higher. This bracket cannot include b, so b will have a different top bracket and will receive a different compact name than a.

Now assume that p is *not* an ancestor of b. In that case the paths from a to p and from a to b diverge at some point. Call this node d. Since d has multiple children, a backedge was added from d to a point at which backedges from only one of the branches were still present. If that point is above a, then the added backedge will bracket a, and either it or a higher backedge will be the top bracket for a, while it could not be the top bracket for b. If the point is below a, then *all* backedges from the branch b is on must have ended below a, so the top bracket for b, whatever it is, must have ended also and so cannot be a bracket of a. In either case, a and b will have different top brackets, and so they will have different compact names. □

3.5 A fast algorithm for cycle equivalence

We can put these observations together into a fast algorithm which makes use of an abstract data type called BracketList to maintain lists of brackets. The following operations are defined on this data type.

create $()$: $BracketList$ — make an empty BracketList structure.

size $(bl : BracketList)$: $integer$ — number of elements in BracketList structure.

push $(bl : BracketList, e : bracket)$: $BracketList$ — push e on top of bl.

top $(bl : BracketList)$: $bracket$ — topmost bracket in bl.

delete $(bl : BracketList, e : bracket)$: $BracketList$ — delete e from bl.

concat $(bl_1, bl_2 : BracketList)$: $BracketList$ — concatenate bl_1 and bl_2.

This abstract data type can be implemented as a record consisting of a doubly-linked list of brackets, a pointer to the last cell of the list, and an integer representing the size of the list. The doubly-linked list permits deletions anywhere in the list. The pointer to the last cell of the list permits fast concatenation of lists by in-place update to the cell. We leave it to the reader to verify that each of the operations of the abstract data type can be implemented in constant time using this concrete representation. The only subtlety is in **delete**. When an edge is pushed onto a bracket list, the edge data structure is updated so it has a pointer to the bracket list cell containing that edge; this permits constant time deletion of an edge from a bracket list.

We use integers to identify cycle equivalence classes. Procedure **new-class** $()$ 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:

- *n.dfsnum* — depth-first search number of node.
- *n.blist* — pointer to node's bracketlist.
- *n.hi* — *dfsnum* of destination node closest to root of any edge originating from a descendant of node n.

The edge data structure saves the equivalence class number and the size of the bracket list when the edge was most recently the topmost bracket of a bracket list. For example, in Figure 3(b), edge z is the topmost bracket for edges c, a and finally b. a is given the same equivalence class number as c because the size of the bracket list at a is the same as it was when z was previously the topmost bracket (at edge c). In contrast, a and b are given different equivalence class numbers. To access the values saved on brackets, each edge structure has the following fields:

- *e.class* — index of edge's cycle equivalence class.
- *e.recentSize* — size of bracket set when e was most recently the topmost edge in a bracket set.
- *e.recentClass* — equivalence class number of tree edge for which e was most recently the topmost bracket.

The edge and node data types can be implemented using records in the obvious way.

Figure 4 gives the pseudocode for computing edge cycle equivalence classes efficiently. It is easy to see that during the depth-first traversal of the undirected graph, the amount of work performed at each node is some constant amount together with work proportional to the number of edges incident at the node. Thus, the algorithm requires $O(E)$ time, where E is the number of edges in the control flow graph.

3.6 Building the program structure tree

Since cycle equivalent edges are totally ordered in the control flow graph by dominance and postdominance, each adjacent pair of edges in this order encloses a canonical SESE region. To find canonical regions, we first compute cycle equivalence classes for edges in $O(E)$ time using the algorithm in Figure 4. Any depth-first traversal of the original control flow graph will visit edges in a given cycle equivalence class in order; during this traversal, entry and exit edges of canonical SESE regions are identified.

Canonical regions can be organized into a *program structure tree* such that a region's parent is the closest containing

```
Procedure CycleEquiv $(G)$
{
1:  perform an undirected depth-first search
2:  for each node $n$ in reverse depth-first order do
3:      /* compute $n.hi$ */ ;
4:      $hi_0 :=$ min $\{t.dfsnum \mid (n,t)$ is a backedge $\}$ ;[4]
5:      $hi_1 :=$ min $\{c.hi \mid c$ is a child of $n\}$ ;
6:      $n.hi :=$ min $\{hi_0, hi_1\}$ ;
7:      $hichild :=$ any child $c$ of $n$ having $c.hi = hi_1$ ;
8:      $hi_2 :=$ min $\{c.hi \mid c$ is a child of $n$ other than $hichild\}$ ;
9:
10:     /* compute bracketlist */
11:     $n.blist :=$ create () ;
12:     for each child $c$ of $n$ do
13:         $n.blist :=$ concat $(c.blist, n.blist)$ ;
14:     endfor
15:     for each capping backedge $d$
            from a descendant of $n$ to $n$ do
16:         delete $(n.blist, d)$ ;
17:     endfor
18:     for each backedge $b$ from a descendant of $n$ to $n$ do
19:         delete $(n.blist, b)$ ;
20:         if $b.class$ undefined then
21:             $b.class :=$ new-class () ;
22:         endif
23:     endfor
24:     for each backedge $e$ from $n$ to an ancestor of $n$ do
25:         push $(n.blist, e)$ ;
26:     endfor
27:     if $hi_2 < hi_0$ then
28:         /* create capping backedge */
29:         $d := (n, node[hi_2])$ ;
30:         push $(n.blist, d)$ ;
31:     endif
32:
33:     /* determine class for edge from $parent(n)$ to $n$ */
34:     if $n$ is not the root of dfs tree then
35:         let $e$ be the tree edge from $parent(n)$ to $n$ ;
36:         $b :=$ top $(n.blist)$ ;
37:         if $b.recentSize \neq$ size $(n.blist)$ then
38:             $b.recentSize :=$ size $(n.blist)$ ;
39:             $b.recentClass :=$ new-class () ;
40:         endif
41:         $e.class := b.recentClass$ ;
42:
43:         /* check for $e, b$ equivalence */
44:         if $b.recentSize = 1$ then
45:             $b.class := e.class$ ;
46:         endif
47:     endif
48: endfor
}
```

Figure 4: The cycle equivalence algorithm[5]

[4]**min** returns infinity (i.e. $N+1$) whenever set is empty.

[5]The code in C is roughly 200 lines long and may be obtained from the authors.

region and its children are all the regions immediately contained within the region. We discover the nesting relationship during the same depth-first traversal that determines canonical regions. The depth-first search keeps track of the most recently entered region (i.e. the *current region*). When a region is first entered, we set its parent to the current region and then update the current region to be the region just entered. When a region is exited, the current region is set to be the exited region's parent. From Theorem 1, it follows that the pushing and popping follows a stack discipline. The topmost SESE region on this stack when DFS reaches the entry node of a SESE region R_1 is the name of the smallest SESE region containing R_1. Once the depth-first traversal is complete, the program structure tree has been built.

4 Empirical properties of the PST

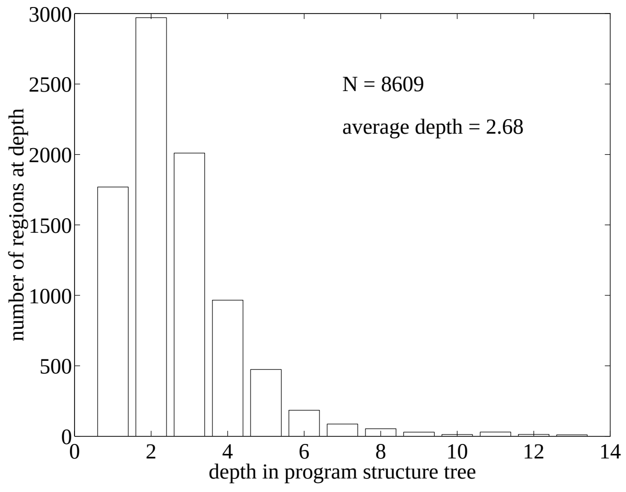
We now present empirical evidence to characterize the properties of the PST. We gathered data from 254 procedures taken from the Perfect Club benchmark suite and the SPEC89 benchmark suite, using Dennis Gannon's Sigma FORTRAN front-end (modified extensively by Mayan Moudgill at Cornell), and a back-end of our own design. The programs are listed below.

suite	program	lines	procedures
Perfect	APS	6105	97
	LGS	2389	34
	TFS	1986	27
	TIS	485	7
SPEC89	dnasa7	1105	17
	doduc	5334	41
	fpppp	2718	14
	matrix300	439	5
	tomcatv	195	1
	linpack	793	11
total		21549	254

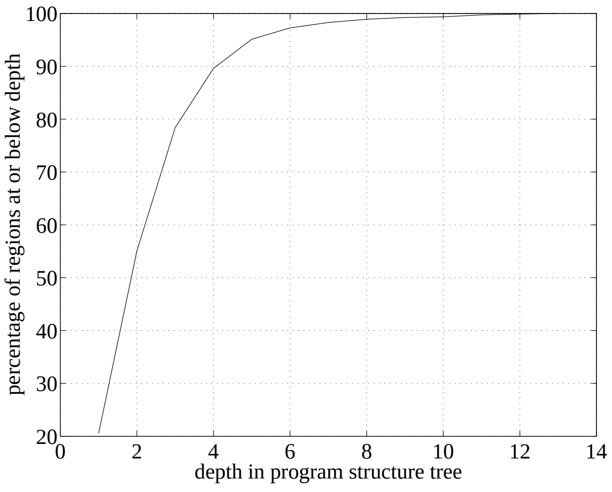
Figure 5(a) presents the distribution of region depth. In the 254 PSTs there are 8609 regions. The maximum depth is 13, and the average depth is 2.68. This agrees with conventional wisdom that typical programs do not contain deeply nested control structures. Figure 5(b) shows the cumulative number of regions at or below each level; from this we see that about 97 percent of all regions have a nesting level of 6 or less.

In Figure 6, we show that as procedures grow larger, the PST also grows in size, but it becomes broader rather than deeper.[6] Figure 6(a) plots each PST's size in number of regions versus procedure size, and we see that the number of regions does grow with procedure size. This indicates that larger procedures have larger opportunities for exploiting structure, as desired. Figure 6(b) shows that the nesting

[6]The 6 largest procedures are omitted from Figures 6 and 9 to avoid compressing the horizontal axis. Their PSTs follow the general trend in each figure.

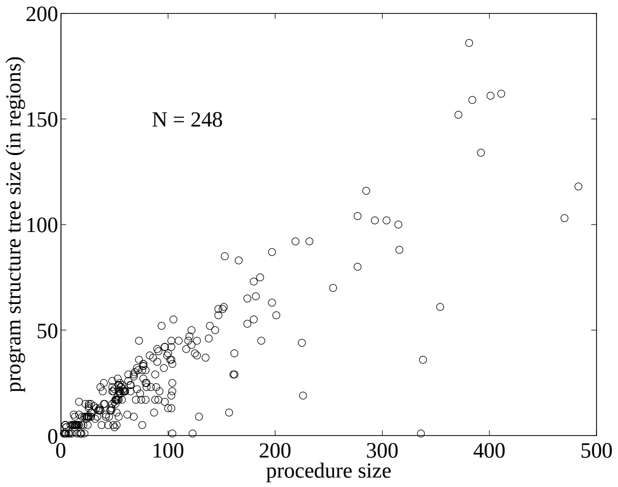


(a) number of regions at each depth in PSTs

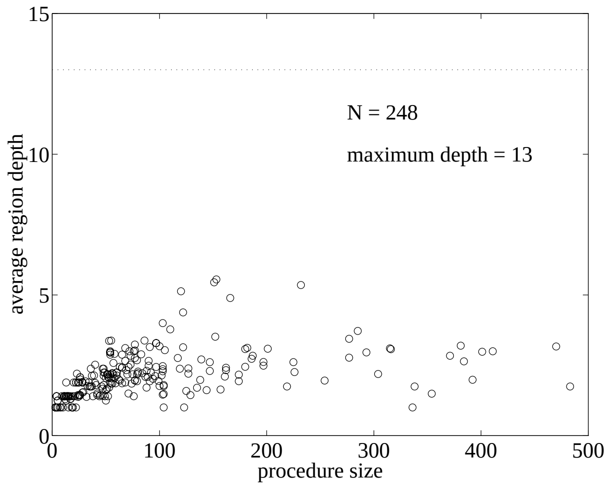


(b) cumulative regions at each depth in PSTs

Figure 5: Analysis of PST depth



(a) PST size versus procedure size



(b) average PST depth versus procedure size

Figure 6: PST size and depth with procedure size

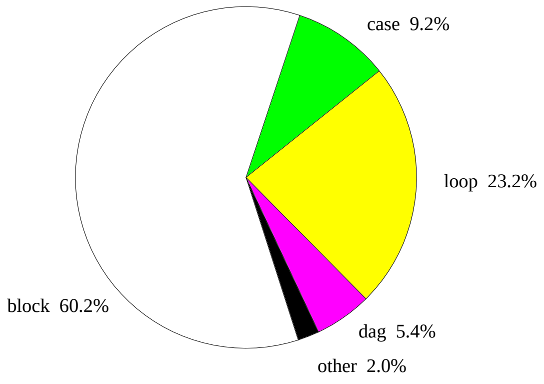


Figure 7: Weighted proportion of regions by kind

depth of structures is independent of procedure size, as expected.

Once SESE regions have been detected, we can further identify the *kind* of structure present in each region. Using a simple pattern-matching pass, we identify each region as being a basic block, a case construct, a loop, a dag, or a cyclic unstructured region. Figure 7 shows the proportion of each kind of region, where each region is weighted by the number of nested maximal SESE regions. For example, an if-then-else has a weight of two since it contains two nested maximal regions. This weighting gives a measure of region size; blocks have unit weight. It is interesting that even this simple heuristic finds considerable structure. In fact, 182 of the 254 procedures are completely structured, and we find considerable structure for the remaining 72 procedures.

These empirical results from standard benchmarks show that real programs contain an abundance of SESE structure that can be exposed quickly by our algorithm. Typical PSTs are flat and broad, not narrow and deep. We now show how the algorithms in this paper and the PST in particular can be used to solve a variety of compilation problems.

5 Control regions in linear time

The first application of our results is to the computation of control regions. This application does not use the PST; rather, it is a reworking of the cycle equivalence algorithm that also motivates the particular definition of single entry single exit regions we have used.

The notion of *control dependence* plays an important role in optimization and parallelization. Intuitively, a node n is control dependent on a node c if c determines whether n is executed. Control dependence is defined formally as follows.

Definition 8 *A node n is* **control dependent** *[FOW87] on node c with direction l if there is a path P from c to n beginning with edge l such that*

1. *n postdominates all nodes other than c on P, and*
2. *if n and c are distinct, n does not postdominate c.*

Control dependence for an *edge* can be defined analogously. Nodes or edges having the same control dependences are in the same control dependence equivalence class, or *control region*. Ferrante, Ottenstein, and Warren first posed the problem of partitioning control flow graph nodes into control regions [FOW87]. Their algorithm used hashing to compute control regions in $O(N)$ expected time, $O(N^2E)$ worst-case time and $O(NE)$ space. These results were improved by Cytron, Ferrante, and Sarkar [CFS90] who gave an $O(EN)$ time, $O(E+N)$ space algorithm for finding control regions. Briefly, their algorithm works by placing all nodes in a single equivalence class and then repeatedly refining the equivalence relation by considering the effect of each control dependence on the existing partition. In the worst-case, the algorithm performs $O(N)$ work for each of $O(E)$ control dependences. The problem with this approach is that control dependence equivalence is defined in terms of the control dependence relation, which has $O(EN)$ size in the worst case. Ball [Bal92] has recognized the need to characterize control dependence equivalences without using control dependence and has developed a linear-time algorithm for computing control dependence equivalences. However, his algorithm works only for reducible graphs and requires computation of both dominators and postdominators. Podgurski has given a linear-time algorithm for *forward* control dependence equivalence, which is a special case of general control dependence equivalence [Pod93].

Using the results of Section 3, we can design an $O(E)$ algorithm to determine control regions of arbitrary flow graphs and which runs faster than just dominator computation, the first step in all previous algorithms for this problem! The key technical result in this section is that control dependence equivalence can be reduced to cycle equivalence.

Theorem 7 *Let S be the strongly connected component constructed by adding the edge* `end` $\rightarrow$ `start` *to a control flow graph G . Nodes a and b in G have the same set of control dependences iff a and b are cycle equivalent in S.*

We leave it to the reader to verify this theorem for the example shown in Figure 1(a). The proof of this theorem is straightforward, if tedious, and can be found in [JPP93]. Unlike the edge cycle equivalence relation, node cycle equivalence is not preserved when edge directions are removed from a graph. Fortunately, a simple construction lets us reduce the problem of finding node cycle equivalence in directed graphs to the problem of edge cycle equivalence in a related directed graph.

Definition 9 *Given a directed graph G, we define a node-expanding transformation $\mathcal{T}$. For each node n in G, there is a pair of nodes n_i and n_o in $\mathcal{T}(G)$, connected with the edge $n_i \rightarrow n_o$; we call this edge the* **representative edge** *for n, denoted as n'. For each edge $n \rightarrow m$ in G, there is a corresponding edge $n_o \rightarrow m_i$ in $\mathcal{T}(G)$.*

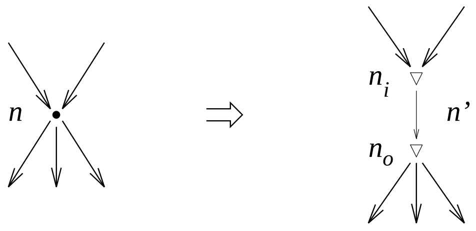


Figure 8: Node expansion

Figure 8 shows the node expansion step pictorially. The following theorem, together with Theorem 7, establishes the reduction of control dependence equivalence to edge cycle equivalence. The proof is obvious and is omitted.

Theorem 8 *Two nodes a and b in a strongly connected component S are* node *cycle equivalent if and only if their representative edges a' and b' are* edge *cycle equivalent in the node-expanded graph, $\mathcal{T}(S)$.*

Therefore, we can use our algorithm for edge cycle equivalence to determine control regions in $O(E)$ time. Our algorithm is asymptotically optimal; in addition, the constant factor is small and the algorithm runs fast in practice. One detail of our *implementation* is worth noting: *we avoid explicitly expanding nodes and undirecting edges*. Instead, we use doubly-linked control flow edges (so that depth-first search can traverse edges in either direction), and we maintain a tuple of information at each control flow node, corresponding to the information that would be stored on the expanded nodes. The resulting code is slightly more complex, but the savings in space and time over working with the explicitly transformed graph are significant. In a related technical report, we have shown that this algorithm runs faster than dominator computation, which is just the first step in all previous algorithms for this problem [JPP93].

6 Applications of the PST

The Program Structure Tree is a tool for enhancing the performance of program analysis algorithms by providing a simple framework for exploiting *global structure*, *local structure*, and *sparsity*. The intuitive idea is the following.

Global structure: The PST is a tree of SESE regions in which nesting structure is made explicit. Moreover, each SESE region is a control flow graph in its own right. Therefore, any global analysis algorithm can be applied unchanged to each SESE region, and the partial results can be combined using the PST to give the global result. This lets us apply analysis algorithms in a divide-and-conquer fashion to the program, which can be a win if the combining of partial results is not overly expensive. For example, suppose we have an $O(N^2)$ algorithm and suppose there are k SESE regions of roughly equal size in the PST of the control flow graph. Provided combining can be done quickly, the cost of the divide-and-conquer approach is approximately $(N/k)^2$ per region, or N^2/k overall, and the algorithm is speeded up by a factor of k. As a concrete example, the *static single assignment* (SSA) form is usually computed using *dominance frontiers* which can be $O(N^2)$ in size [CFR+91]. We show that using the PST, SSA computation can be performed separately in each SESE region. Since the size of a SESE region is roughly independent of program size (Figure 9) and there is *no* combining of partial results to be done in this problem, PST-based exploitation of nesting structure is a win.

Local structure: The PST lets us tailor analysis algorithms to the structure of each SESE region. Figure 7 shows that in practice, most SESE regions are basic blocks, conditionals, DAGs and loops; therefore, fast algorithms can be used for these regions even if other regions in the PST are unstructured or even irreducible. One way to view this is that the PST lets us 'localize' the effect of lack of structure into SESE regions which do not affect analysis of other regions.

Sparsity: In many analysis problems, the solution is determined by a small subset of the SESE regions in the PST; the other regions do not contribute to the solution and need not be analyzed. For example, in converting a program to SSA form, we show that ϕ-function placement for a variable x can be solved completely by analyzing only those regions that contain an assignment to x. This lets us ignore the vast majority of SESE regions, as we show experimentally.

We illustrate these points by discussing how the PST can be used to speed up algorithms for two problems — computing the static single assignment form and performing data flow analysis. In particular, our experimental results highlight the importance of exploiting sparsity.

6.1 Using the PST in conversion to SSA form

Translation into SSA form requires the introduction of ϕ-*functions* at some merge points in the control flow graph. Cytron *et al* [CFR+91] showed that a ϕ-function is needed at a merge if it is the first point in common on two paths from distinct definitions of a variable v to a use of v. They characterized this set of merges in terms of a property called the *dominance frontier*. Briefly, a merge m is in the dominance frontier of a node n, $DF(n)$, if n dominates a predecessor of m but does not dominate m. Extending dominance frontiers to sets, $DF(S) = \cup_{s \in S} DF(s)$. For a variable v defined at nodes in the set V, Cytron *et al* showed that the set of merges needing ϕ-functions for v is exactly the *iterated* dominance frontier, $DF^+(V)$, which is the limit of the sequence $DF_{i+1} = DF(V \cup DF_i)$, where $DF_1 = DF(V)$. The computation of $DF^+(V)$ is performed with a worklist algorithm. The size of the dominance frontier of a node is $O(N^2)$ in the worst case.

Our algorithm uses the nesting structure in the PST to avoid computing the entire dominance frontier for each node. The key theorem is the following one.

Theorem 9 *If a merge node needs a ϕ-function for variable v, then it is in the iterated dominance frontier of some assignment to v in the same SESE region as the merge node.*

We omit the proof and describe only the intuition. First, consider dominance frontiers. If the merge is the first node in common on two paths from distinct definitions of v, then both definitions cannot be outside the region containing the merge, since then the two paths must join prior to entering the merge's region. Likewise, both definitions cannot be in the same region nested within the merge's region, since the two paths would join prior to exiting this nested region. Therefore, a merge that is in the dominance frontier of two assignments to v must be in the same SESE region as one of them. By induction on the definition of *iterated* dominance frontiers, the result is proved for iterated dominance frontiers in general. Note that this implies that any region containing no definitions of v needs no ϕ-functions.

We use this result to exploit both global structure and sparsity in the PST. Instead of computing dominance frontiers for an entire procedure, we compute dominance frontiers for each SESE region separately. This can be advantageous — for example, the size of dominance frontiers for nested repeat-until loops reaches the worst-case bound of $O(N^2)$ [CFR+91]. When we exploit nesting structure using the PST, *each loop is a SESE region whose dominance frontiers are computed independently*, thereby avoiding the quadratic blowup. This is an example that illustrates the exploitation of global structure using the PST. To exploit sparsity, we note that SESE regions that do not contain an assignment to the variable can be omitted from the analysis. Putting these observations together gives us the following algorithm for enhancing the performance of SSA algorithms.

Algorithm for ϕ-placement:

Build the program structure tree.
For each variable v, do the following.

1. In the PST, mark every region containing an assignment to v.
2. For each region, collapse immediately nested regions into single 'statements' as follows: if that region contains a definition of the variable, treat the region as a definition of the variable; otherwise, treat the region as a NO-OP. From Theorem 9, it follows that collapsing nested regions as described maintains the path properties that determine where ϕ-functions are needed.
3. Apply any algorithm for finding the SSA form to each marked region, treating the entry point of the region as a definition and the exit as a use of the variable.

By maintaining a list of definitions for each variable, we can perform the marking step in time proportional to the number of regions marked. Figure 10 shows the fraction of SESE regions examined when placing ϕ-functions for 5072 variables. We see that for most variables, only a small fraction of SESE regions are examined. Seventy percent of variables required examining less than one-fifth of the regions. In Step 3, it is possible to exploit local structure and use different SSA algorithms in each region if that is desired. For example, it is trivial to convert if-then-else and loop structures into SSA form. Figure 7 suggests it might be worth doing this type of algorithm specialization. Moreover, the PST can even be distributed across the local memories of a parallel machine, and computations in SESE regions can be performed in parallel. Given the overheads of parallel computation on current machines, this approach is unlikely to yield much speed-up, but the principle is clear — the PST can be used to exploit parallelism in compilation since it tells us how to 'divide' the work and how to 'combine' partial results.

The divide-and-conquer strategy works particularly well in this problem because no 'combining' of individual region solutions is needed to generate the solution for the entire procedure. Next, we discuss dataflow analysis, a problem in which region solutions must be combined to yield the solution for the entire procedure.

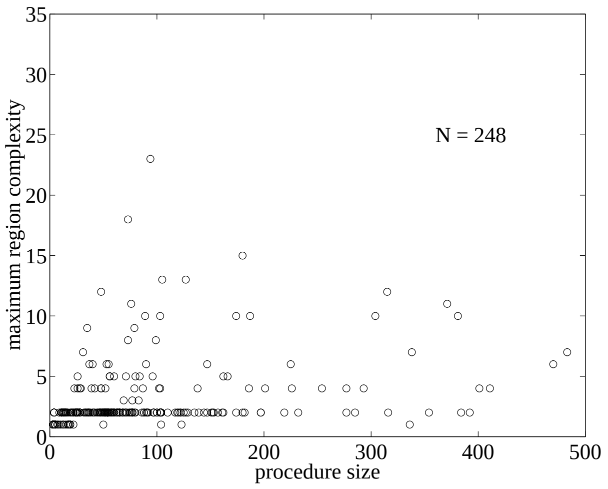


Figure 9: Maximum region size versus procedure size

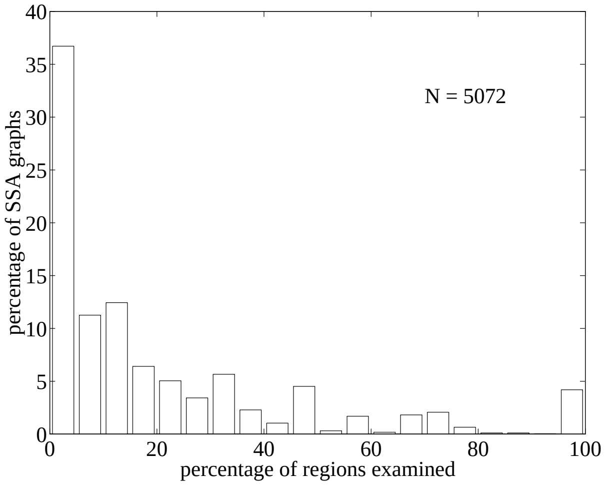


Figure 10: Percentage of regions examined while placing ϕ-functions

6.2 Using the PST in data flow analysis

Solution techniques for monotone data flow analysis problems are classified into iterative methods and elimination methods [Ken81, RP86]. We show discuss how the PST can be used with either class of methods.

Exploiting global and local structure: Elimination methods exploit nested program structure to solve data flow equations efficiently. Given some hierarchical decomposition of program structure, analysis is performed in two phases. In the first phase, local information is computed for increasingly larger regions of the program; at each stage only the information from nested regions is taken into consideration. In the second phase, global information is propagated to increasingly smaller regions. The classic approach to elimination algorithms uses an interval decomposition of the program [AC76].

The PST can be used as the hierarchical decomposition for solving data flow systems via the elimination method: in the first phase, local information is computed for each SESE region in bottom-up order in the PST, and then global information is propagated from larger to smaller SESE regions during a top-down traversal of the PST. In both phases, we need some algorithm to collect or propagate information within a SESE region. As discussed in Section 4, most regions are simple constructs such as blocks, *if-then* or *loop* constructs; these regions may be processed quickly using structure-based methods [Ken81]. What about the remaining unstructured regions? An important aspect of the PST is that it is compatible with methods based on intervals. In particular, we have the following theorem whose proof is straightforward.

Theorem 10 *If a control flow graph G is reducible, then all SESE regions of G are reducible.*

Therefore, if the original graph is reducible, the (few, small) unstructured SESE regions in the PST can be analyzed using interval methods. Finally, for irreducible regions, we can fall back on a general iterative method, which is similar in spirit to so-called 'hybrid' algorithms [Zad84, HDT87, MR90]. It is interesting to note that Graham and Wegman exploited single-exit intervals to speed up elimination-based data flow analysis [GW76].

Exploiting sparsity: Recent work on speeding up data flow analysis has focused on solving individual instances of data flow problems, such as finding the availability of $x + y$, as opposed to analyzing a property for all variables or expressions simultaneously as is done in the traditional bit-vector approach. In this case, much of the control flow graph does not contribute (i.e. modify or use) to the solution. *Sparse methods* of data flow analysis attempt to avoid propagating information through regions of the program where the data flow values are not modified.

Our approach to exploiting sparsity using the PST is to bypass SESE regions having only identity transfer functions. It is easy to show that bypassing such "transparent" regions does not effect the global data flow solution. Given an data flow problem instance, we build a *quick propagation graph (QPG)*, which is much smaller than the control flow graph, and then solve the data flow system using this graph. The solution in the QPG can then be projected back into the control flow graph. The nodes in a QPG are a subset of the control flow graph nodes, and each edge in a QPG is denoted by a pair of control flow edges (e_1, e_2) such that either e_1 and e_2 are the same edge, or (e_1, e_2) encloses a SESE region. Therefore, the QPG edge connects the source of e_1 to the destination of e_2. QPGs are constructed so that each edge bypasses a maximal SESE region having only identity transfer functions. (Optimizations to the QPG that allow additional forms of bypassing and special treatment of constant transfer functions are discussed in Johnson's dissertation.)

Once the quick propagation graph is built, the data flow system is solved using this graph, thereby avoiding transparent regions altogether. Since bypassing is performed on the basis of SESE regions, and since these regions are also the basis for exploiting structure using an elimination method, *use of the PST allows structure and sparsity to be exploited simultaneously.* Of course, nothing precludes the use of an iterative method for the entire QPG. Once the solution in the QPG is obtained, it is a simple matter to transfer this solution to the CFG as explained below.

Algorithm for PST-based data flow analysis:

1. Mark SESE regions containing a non-identity transfer function. This is done by starting at the leaf nodes (i.e. basic blocks) having statements with non-identity transfer functions and then marking all ancestors in the PST.
2. Construct the QPG by traversing the CFG, bypassing any unmarked SESE regions as explained above.
3. In the QPG, solve the data flow system using any solution method.
4. Transfer the solution from the QPG to the CFG as follows. Every edge in the CFG is either present in the QPG or it is part of a transparent SESE region (e_1, e_2) bypassed in the construction of the QPG. In the first case, the data flow solution on the corresponding QPG edge is transferred to the CFG edge. In the second case, the data flow solution on edge e_1 (or e_2) in the QPG is transferred to the CFG edge.

Note that the marking step can be done in time proportional to the number of marked regions if we know the location of the non-identity transfer functions. For common optimizations, the non-identity transfer functions can be found by maintaining a list of definitions and uses for each variable. The total time required to build a QPG is proportional to the size of the QPG plus the number of marked PST regions. In the worst case, all PST regions are marked, no regions are bypassed, and the QPG is simply the original CFG.

As we have shown in Section 4, PSTs tend to be broad and shallow. Therefore, if the number of leaf nodes containing non-identity transfer functions is small, then the total number of regions which cannot be bypassed will be small. Preliminary studies show that the QPG is usually quite small compared to the original CFG, averaging less that 10% the size of the (statement-level) CFG. Since QPGs are often so small relative to the size of the CFG, it is a significant savings that our algorithm does not examine transparent regions. In a previous paper, we discussed a representation of dependences called the *dependence flow graph*(DFG) [JP93]. Intuitively, the DFG is a set of 'basis' graphs from which we can construct the QPG for a given data flow problem. For lack of space, we postpone discussion of this connection.

In principle, the PST (or QPG) can be used to perform data flow analysis in parallel — as is standard with divide and conquer algorithms, we work on leaf regions in parallel, and work on an interior node of the PST (or QPG) when all its children have been processed. We refer the interested reader to related work by Gupta, Pollack and Soffa [GPS90] who use the SESE decomposition of programs in a structured programming language to perform data flow analysis in parallel. Note that our definition of SESE regions is stronger than theirs since we require unique entry and exit edges, whereas they allow multiple edges to the entry node from outside the region, as well as out of the exit node. As in the case of SSA computation, parallel data flow analysis is likely to be a whimsical idea unless communication latencies on parallel machines are reduced significantly.

6.3 Discussion

The PST can be used to design divide-and-conquer style algorithms for a surprising variety of problems. For example, it is not difficult to design such an algorithm for computing the dominator tree of a control flow graph — first, build the dominator tree of each SESE region, and then piece together the local trees using global structure (nesting) information in the PST. Such an approach might lead to fast *incremental* algorithms for analysis problems since the PST can be used to isolate regions of the graph where information must be recomputed. The PST is also useful in generating code for dataflow machines from programs in a language like FORTRAN or C since it exposes SESE regions which dataflow edges can potentially bypass [BJP91, BMO90].

There is an enormous body of work on elimination and iteration algorithms, and we refer the reader to surveys by Ryder and Paull [RP86], and by Kennedy [Ken81]. Tarjan and Valdes use a hierarchical representation of SESE regions of a different kind to do elimination [Val78, TV80]. Sparsity was highlighted by Choi, Cytron, and Ferrante [CCF91], and by Dhamdhere, Rosen, and Zadeck [DRZ92]. Choi *et al* extend the SSA form to build sparse evaluation graphs (SEGs); these graphs also bypass uninteresting regions of the control flow graph and in general will be smaller than our quick propagation graphs. However, they are more costly to build and it is unclear how to exploit both sparsity and structure using SEGs, since their edges cross interval (or SESE region) boundaries in an ad hoc manner. Recently, Cytron and Ferrante [CF93] have improved the time for placing ϕ-functions (needed to build SSA form and SEGs) to $O(E\alpha(E))$ time; Sreedhar and Gao [SG94] have a linear-time algorithm for ϕ-function placement. It would be interesting to compare the performance of these algorithms to the performance of a PST based algorithm that used the dominance frontier algorithm [CFR$^+$91] selectively in the few, small unstructured SESE regions in the PSTs of typical programs.

7 Conclusions

The program structure tree (PST) is a hierarchical representation of program structure in which nodes represent single entry single exit (SESE) regions and edges represent region nesting. The PST is defined for arbitrary flow graphs, even irreducible ones. We showed that finding SESE regions is equivalent to solving the naturally stated graph problem of cycle equivalence: edges are equivalent iff each cycle in the graph contains all or none of the edges in an equivalence class. In this paper, we discussed an $O(E)$ algorithm for the cycle equivalence problem and used it to compute the PST of a control flow graph in $O(E)$ time.

We presented experimental evidence that real programs contain abundant SESE regions organized into broad, shallow PSTs; even the worst unstructured portions of procedures contain nested structure and comprise only a small fraction of the total procedure size. Intuitively, the PST enables us to isolate the effect of lack of structure into small SESE regions, thereby letting us exploit structure globally.

Our results have many applications. We showed that the problem of determining control regions, which is needed in global code scheduling for example, can be solved in $O(E)$ time using the cycle equivalence algorithm.[7] The recursive structure of the PST makes it possible to design divide-and-conquer style algorithms for control flow and data flow problems, exploiting global structure, local structure, and sparsity.

We conclude that single entry single exit regions and their nesting relationship provide a simple, intuitive, and inexpensive approach to representing and exploiting hierarchical program structure based on control dependence equivalence.

Acknowledgments: We would like to thank Bob Tarjan for his extensive comments on an earlier version of this paper, and for bringing his work with Jacobo Valdes to our attention. Thanks also to Mayan Moudgill who wrote and maintained some of the software used to gather the experimental results. Discussions with Dexter Kozen and Eva Tardos helped us simplify the presentation of the results in this paper. Finally, Micah Beck, Wei Li, and Paul Stodghill gave us extensive feedback throughout this research.

[7]The PST can be used to give a linear time and space factorization of control dependences that usually returns control dependence sets in time proportional to their size. The problem of providing such a factorization that always returns control dependence sets in proportional time remains open.

References

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[Bal92] Thomas Ball. What's in a region? -or- computing control dependence regions in linear time and space. Technical Report 1108, University of Wisconsin – Madison, Computer Sciences Department, September 1992. To appear in LOPLAS.

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

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, White Plains, New York, June 20–22, 1990.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, Orlando, Florida, January 21–23, 1991.

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

[CFR$^+$91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CFS90] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Compact representations for control dependence. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 337–351, White Plains, New York, June 20–22, 1990.

[DRZ92] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, San Francisco, California, June 17-19, 1992.

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

[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.

[GW76] S. Graham and M. Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[HDT87] S. Horwitz, A. Demers, and T. Teitelbaum. An efficient general iterative algorithm for data-flow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[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.

[JPP93] Richard Johnson, David Pearson, and Keshav Pingali. Finding regions fast: Single entry single exit and control regions in linear time. Technical Report 93-1365, Department of Computer Science, Cornell University, July 1993.

[Kas75] V. N. Kas'janov. Distinguishing hammocks in a directed graph. *Soviet Math. Doklady*, 16(5):448–450, 1975.

[Ken81] Ken Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Application*, chapter 1, pages 5–54. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[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.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, San Francisco, California, January 1990.

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

[RP86] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[SG94] Vugranam C. Sreedhar and Guang R. Gao. Computing ϕ-nodes in linear time using DJ-graphs. Technical Report ACAPS Technical Memo 75, McGill University School of Computer Science, January 1994.

[TV80] Robert E. Tarjan and Jacobo Valdes. Prime subprogram parsing of a program. In *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 95–105, Las Vegas, Nevada, January 28–30, 1980.

[Val78] Jacobo Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. PhD thesis, Stanford University, December 1978. Report STAN-CS-78-682.

[Zad84] F. Kenneth Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 132–143, Montreal, Canada, June 17–22, 1984.