What's in a Region? -or- Computing Control Dependence Regions in Linear Time and Space

Thomas Ball

Technical Report #1108

September 1992

What's in a Region?

- or -

Computing Control Dependence Regions in Linear Time and Space

THOMAS BALL*
tom@cs.wisc.edu
Computer Sciences Department
University of Wisconsin – Madison
1210 West Dayton Street
Madison, Wisconsin 53706 USA
608-262-6612

August 19, 1992

Abstract

Regions of control dependence identify the instructions in a program that execute under the same control conditions. They have a variety of applications in parallelizing and optimizing compilers. Two vertices in a control flow graph (which may represent instructions or basic blocks in a program) are in the same region if they have the same set of control dependence predecessors. The best known algorithm for computing regions takes $O(V \times E)$ time, where V and E are the number of vertices and edges in the control flow graph, respectively. We present algorithms for finding regions in $O(V+E)$ time and $O(V+E)$ space, without using control dependence. These algorithms are based on alternative definitions of regions, which are easier to reason with than the definitions based on control dependence.

1. INTRODUCTION

Regions of control dependence identify the instructions in a program that execute under the same control conditions. They have a variety of applications in parallelizing and optimizing compilers [3, 5] and other systems [8]. For example, regions can be used for identifying code that can be executed in parallel, and for global instruction scheduling by identifying code that may be moved between basic blocks [2].

Two vertices in a control flow graph (which may represent instructions or basic blocks in a program) are in the same region if they have the same set of control dependence predecessors. Two queries regarding regions are useful: (1) are vertices v and w in the same region?; (2) what vertices are in the same region as vertex v? By determining the partioning of vertices that regions induce, both questions can be answered efficiently. The best known algorithm for computing (the partitioning induced by) regions takes $O(V \times E)$ time, where V and E are the number of vertices and edges in the control flow graph, respectively [4]. This is because the algorithm examines each control dependence once, and there can be $O(V \times E)$ control dependences in the worst-case, even for acyclic control flow graphs. This algorithm uses $O(V+E)$ space.

This paper presents algorithms for finding regions in linear time and space, *without using control dependence*. These algorithms are based on alternative definitions of regions:

* This work was supported in part by the National Science Foundation under grant CCR-8958530, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from Xerox and 3M.

- Vertices v and w are in the same *weak region* iff for any complete control flow path,[1] v and w are both in the path or are both absent from the path. Weak regions are equivalent to the control dependence regions that arise from forward (loop-independent) control dependences [3].

- Vertices v and w are in the same *strong region* iff v and w occur the same number of times in any complete control flow path. Strong regions are equivalent to the control dependence regions that arise from full control dependences.

We present algorithms to find weak regions in $O(V+E)$ time and $O(V+E)$ space for all control flow graphs, and to find strong regions in $O(V+E)$ time and $O(V+E)$ space for reducible control flow graphs. We identify a property of weak regions that allows them to be computed in a single pass over the postdominator tree of the control flow graph, in conjunction with queries on the dominator tree. Strong regions can be identified by loop analysis in conjunction with weak region identification, and for a certain class of reducible graphs can be computed without the aid of loop analysis.

The running time for the algorithms includes the time needed to construct the postdominator and dominator trees, and to perform the loop analysis. However, as this information is commonly computed for other purposes by program transformation systems, in the context of such systems it is free. The running time for the algorithms, not considering the time needed to compute this information, is still $O(V+E)$.

This paper makes two major contributions:

(1) The first linear time algorithm for computing regions of control dependence. Previous algorithms have poor worst-case performance. However, such algorithms typically may perform well in practice because situations where $O(V \times E)$ control dependences arise are rare. Nevertheless, our algorithm performs well in all cases and is simple to implement.

(2) A new characterization of regions based on execution frequency of vertices in control flow paths. This characterization is equivalent to that based on control dependence. The declarative nature of this definition makes it easier to reason with than the definition based on control dependence.

The paper is organized as follows: Section 2 defines the control flow graph, describes some applications of regions, and reviews the concepts of domination and postdomination. Section 3 shows how weak regions can be efficiently computed using the postdominator and dominator trees. Section 4 augments the algorithm for weak regions to compute strong regions. Section 5 shows that our characterization of regions is equivalent to that based on control dependence.

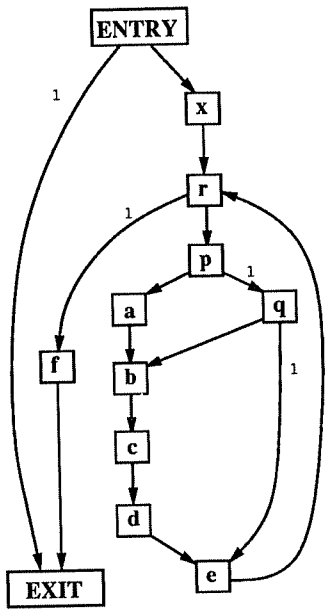
2. BACKGROUND

2.1. The control flow graph

The control flow graph is a directed graph, rooted at the *ENTRY* vertex. Vertices in the control flow graph represent the instructions or basic blocks in a program. There is a distinguished *EXIT* vertex (with no successors) and an edge from *ENTRY* to *EXIT*. Every vertex in the graph is reachable from *ENTRY* and *EXIT* is reachable from every vertex. The outgoing edges of each vertex are uniquely labelled. A *complete path* in the control flow graph is a directed path from *ENTRY* to *EXIT*. Each complete path represents a possible program execution.

---

[1] A *complete control flow path* is a path from the entry point of the control flow graph to the exit point.



Weak Regions

{ ENTRY , EXIT }

{ x , r , f } { p , e } { b , c , d }

{ a } { q }

Strong Regions

{ ENTRY , EXIT }

{ x , f } { p , e } { b , c , d }

{ a } { q } { r }

Figure 1. A control flow graph with weak and strong regions identified.

2.2. Applications of regions

The Introduction presented the definitions of weak and strong regions. Figure 1 presents an example control flow graph, with weak and strong regions identified. We describe the application of strong regions to code scheduling and profiling.

Code schedulers typically reorder the instructions within a basic block to improve program performance (provided, of course, that data dependences are respected). Strong regions identify situations where instructions can be moved across basic block boundaries without code duplication and without incurring a penalty in the number of instructions executed for any complete path. For example, in Figure 1, if we wish to move an instruction from vertex *e* to vertex *d*, the same instruction may have to be copied to vertex *q* to ensure correctness. However, this duplication may increase the cost of a loop iteration that includes both vertices *q* and *d* (as each vertex now includes an extra instruction). Moving an instruction from vertex *e* to vertex *r* does not require duplication but introduces an extra instruction on the path *ENTRY*→*x*→*r*→*f*→*EXIT*. Moving an instruction from vertex *e* to vertex *p*, which is in the same strong region as *e*, requires neither code duplication nor incurs an instruction count penalty.

Regions can also be used to profile programs efficiently. The problem of vertex profiling is to instrument the control flow graph with counting code so that the number of times each vertex (basic block) appears in an execution can be determined. A naive solution is to associate a counter with every vertex. A better method is to allocate one counter to every strong region. By the definition of strong region, the count for all the vertices in the same strong region must be the same. In Figure 1, only six counters are needed.

2.3. Dominators and postdominators

The computation of weak and strong regions rely on the concepts of domination and postdomination in the control flow graph. Let v and w be vertices in a control flow graph. Vertex v *dominates* vertex w, denoted by v **dom** w, if $v \neq w$ and v is on every path from *ENTRY* to w.[2] Vertex v *immediately dominates* w, denoted by v **idom** w, if v **dom** w and there is no vertex z such that v **dom** z **dom** w. Vertex v *postdominates* vertex w, denoted by v **pd** w, if $v \neq w$ and v is on every path from w to *EXIT*. Immediate postdominance (v **ipd** w) is defined similarly to immediate dominance. Postdominance can be defined as dominance in the reverse control flow graph, in which the direction of edges is reversed and the *ENTRY* and *EXIT* vertices are interchanged.

The dominator (or postdominator) relation can be represented as a tree where v is the parent of w iff v **idom** w (or v **ipd** w) and v is a proper ancestor of w iff v **dom** w (or v **pd** w). Figure 2 presents the dominator and postdominator trees for the example control flow graph. Dominator and postdominator trees can be computed in $O(E)$ time [7]. Each tree requires $O(V)$ space.

3. WEAK REGIONS

Vertices v and w are in the same weak region of a control flow graph G iff for any complete path in G, v and w are both in the path or are both absent from the path. It is straightforward to show that distinct vertices v and w are in the same weak region iff (v **dom** w and w **pd** v) or (w **dom** v and v **pd** w).[3] In Figure 1, p **dom** e and e **pd** p, so vertices p and e are in the same weak region. Considering the vertices p and q, q does not postdominate p and p does not postdominate q, so this pair of vertices cannot be in the same weak region.

Weak regions partition the vertex set of the control flow graph. Given a vertex v, we would like to determine the other vertices in the same weak region as v. Figure 2 presents the dominator and postdominator trees of the control flow graph in Figure 1 with the weak regions identified in both trees with shading. The key observation about weak regions that allows a linear time algorithm is that for any control flow graph, the vertices of each weak region form a chain in the postdominator tree[4] that is the reverse of a chain in the dominator tree. Computing weak regions reduces to the problem of computing those chains in one tree that are the reverse of chains in the other tree. This can be accomplished easily by a depth-first search of either tree. We first prove the chain property of weak regions and then describe the depth-first search algorithm. The chain property of weak regions relies on the following lemma:

LEMMA (1). Given any control flow graph, if a **dom** c and c **pd** a then a **dom** b **dom** $c \Leftrightarrow c$ **pd** b **pd** a.

PROOF.

($\Rightarrow$) Suppose that a **dom** b **dom** c. This implies that every path from a to c includes b. Since c **pd** a, every path from a to *EXIT* includes c, so every path from a to *EXIT* must also include b (b **pd** a). Suppose that there is a c-free path from b to *EXIT* (not c **pd** b). Since every path from a to c includes b (and a, b, and c are pairwise unique), there must be a c-free path from a to b. The above two facts imply that there is a c-free path from a to *EXIT*, contradicting an initial assumption. Therefore, c **pd** b.

---

[2]This differs slightly from the usual definition of dominance, which is reflexive.

[3]This definition is identical to Bernstein and Rodeh's notion of *equivalent* vertices [2]. However, they use forward control dependences to discover the classes of equivalent vertices, which is not as efficient as the method given here.

[4]A *chain* in a tree T is a sequence of vertices $(v_1, \cdots, v_n)$ such that for all i, v_i is a parent of v_{i+1} in T.

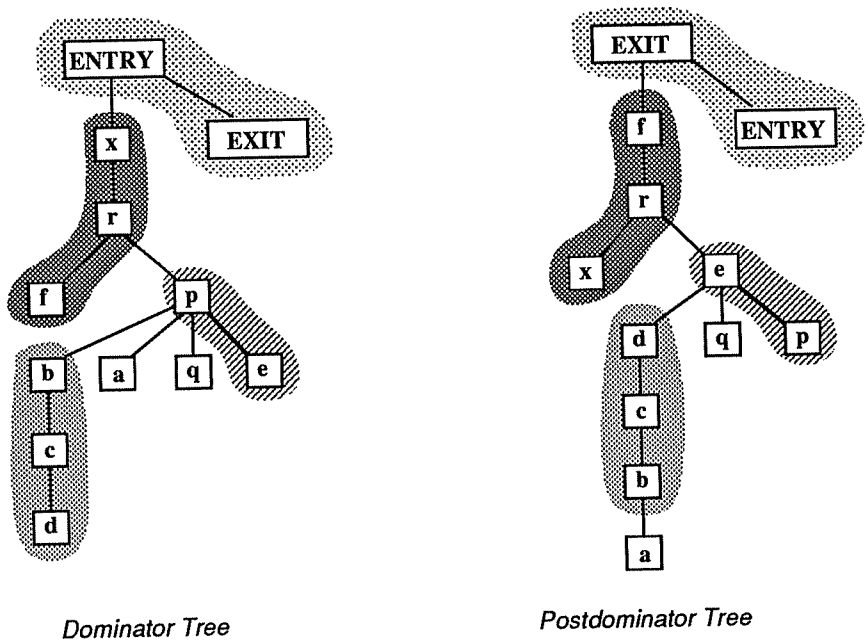


Figure 2. The dominator and postdominator trees of the control flow graph from Figure 1, with weak regions identified. A vertex that is not shaded is in a weak region by itself. Weak regions partition the postdominator and dominator trees into chains that are the reverse of one another.

($\Leftarrow$) Because dominance and postdominance are symmetric, the proof is symmetric to ($\Rightarrow$). □

THEOREM (1). For any control flow graph, each weak region forms a chain in the postdominator tree that is the reverse of a chain in the dominator tree.

PROOF. For every pair of distinct vertices (v,w) from weak region R, either (v **dom** w and w **pd** v) or (w **dom** v and v **pd** w). Since every pair of vertices from weak region R is related by postdominance, there is a chain in the postdominator tree that contains every vertex in R. Let $(v_1, \cdots, v_n)$ be the smallest chain in the postdominator tree that contains every vertex in R.

We first show every vertex in this chain is in R. Since v_1 and v_n are members of R, and v_1 **pd** v_n, it follows that v_n **dom** v_1. Consider any vertex v_i, where $1<i<n$. Since v_1 **pd** v_i **pd** v_n and v_n **dom** v_1, lemma (1) implies that v_n **dom** v_i **dom** v_1. Therefore, v_i must be in R.

We now argue that for all $i<n$, v_{i+1} **idom** v_i. Since v_{i+1} and v_i are in weak region R and v_i **ipd** v_{i+1}, it follows that v_{i+1} **dom** v_i. If there is a vertex z such that v_{i+1} **dom** z **dom** v_i, then lemma (1) implies that v_i **pd** z **pd** v_{i+1}, which contradicts v_i **ipd** v_{i+1}. Therefore, v_{i+1} **idom** v_i. □

As in [4], we use the following data structures to represent regions:

WREGION(v)	the weak region number associated with vertex v	
WHEAD(R)	the first vertex in weak region R	(*i.e.*, lowest in postdominator tree)
WTAIL(R)	the last vertex in weak region R	(*i.e.*, highest in postdominator tree)
WNEXT(v)	the vertex after v in WREGION(v)	(*i.e.*, WNEXT(v) **ipd** v)
WPREV(v)	the vertex before v in WREGION(v)	(*i.e.*, v **ipd** WPREV(v))

For each vertex in the control flow graph, three pieces of information are maintained (WREGION, WNEXT, and WPREV), and for each weak region, two pieces of information are needed (WHEAD and WTAIL). Since there can be at most V weak regions, the size of these data structures is $O(V)$.

Figure 3 presents the depth-first search algorithm for computing weak regions. The global variable *region_num* keeps track of the number of weak regions (chains) found so far. The depth-first search is done on the postdominator tree (although because of symmetry, it could just as easily be done on the dominator tree). *EXIT* is the tail of the first weak region (lines [3] and [4]).

The procedure DFS finds chains in the postdominator tree that are the reverse of chains in the dominator tree. When examining a child w of vertex v in the postdominator tree (line [7]), the algorithm checks if w is the parent of v in the dominator tree (line [8]). If so, then v and w are in the same weak region (lines [9-10]). If not, then vertices v and w cannot occupy the same weak region (lines [11-14]). A new weak region is created and w is the tail of this region (the depth-first search builds weak regions in reverse order).

4. STRONG REGIONS

Vertices v and w are in the same strong region of a control flow graph G iff for any complete path in G, v and w occur the same number of times in the path. Any vertices that are in the same strong region are necessarily in the same weak region. For acyclic control flow graphs, weak regions and strong regions are equivalent. However, for cyclic control flow graphs, two vertices may be in the same weak region but in

```
                                    DFS(v : vertex , num : integer )
                                    begin
                                    [6]  WREGION(v), WPREV(v) := num, nil;
                                    [7]  for each vertex w in PDOM(v).children do
                                    [8]     if DOM(v).parent = w then
                                    [9]        WPREV(v), WNEXT(w) := w, v;
                                    [10]       DFS(w,num);
                                           else
                                    [11]       WHEAD(num) := v;
                                    [12]       region_num := region_num+1;
                                    [13]       WTAIL(region_num), WNEXT(w) := w, nil;
                                    [14]       DFS(w,region_num);
                                        fi
                                     do
                                    end
begin
[1]  compute postdominator and dominator trees;
[2]  region_num := 1;
[3]  WTAIL(region_num) := EXIT;
[4]  WNEXT(EXIT) := nil;
[5]  DFS(EXIT, 1);
end
```

Figure 3. Computing weak regions with the postdominator and dominator trees. PDOM(v).children is a list of v's children in the postdominator tree and DOM(v).parent is v's parent in the dominator tree.

different strong regions. For example, in Figure 1, vertices x and r are in the same weak region, but are not in the same strong region, since r is in a loop that does not contain x. Vertices x and f are in the same strong region.

In order to compute strong regions (without using control dependence) we need to reason about loops, in addition to domination and postdomination. Stated informally, if two vertices are in different loops then they must be in different strong regions. However, just because two vertices are in the same loop and weak region does not imply that they are in the same strong region. There may be a cycle that contains one vertex but not the other. For example, in Figure 4, although vertices a, b, and c are in the same loop and the same weak region, vertex c is not in the same strong region as vertices a and b. Given this intuition, it is fairly straightforward to see that strong regions can be characterized as follows:

> Distinct vertices v and w are in the same strong region iff ((v **dom** w and w **pd** v) or (w **dom** v and v **pd** w)) and (v is in every cycle containing w) and (w is in every cycle containing v).

This section describes how to compute strong regions efficiently for reducible control flow graphs, using loop analysis in conjunction with weak region identification. Section 4.1 reviews the concepts of reducibility and natural loop analysis. Section 4.2 shows how to compute strong regions efficiently. Section 4.3 presents a class of reducible control flow graphs for which no loop analysis is needed to identify strong regions.

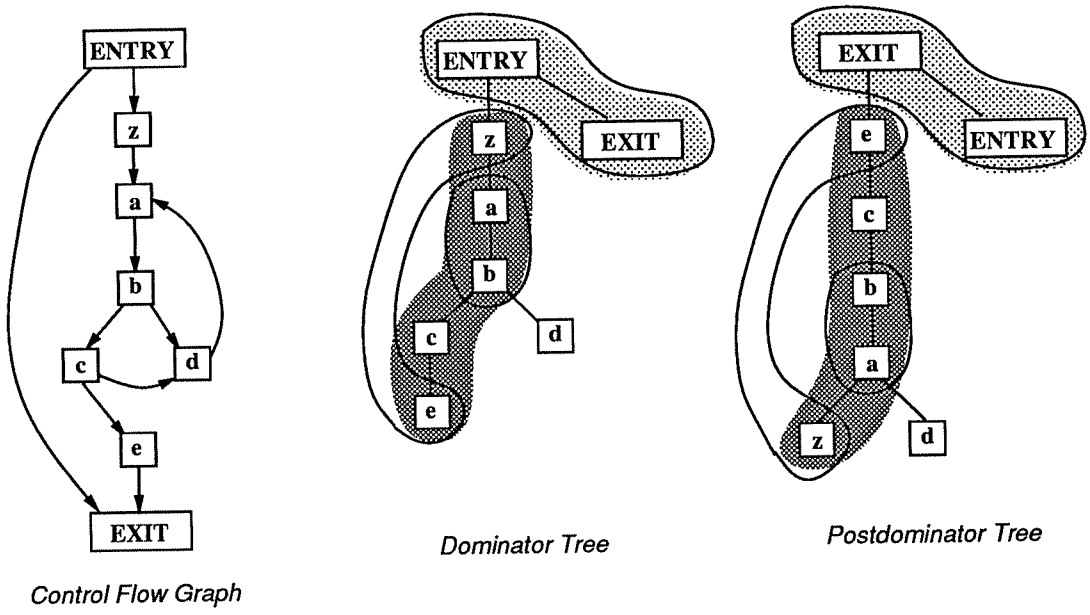


Figure 4. A control flow graph that shows that loops do not necessarily partition weak regions into strong regions. Weak regions are identified by shading in the dominator and postdominator trees, while strong regions are identified by outlines. A vertex that is not shaded (outlined) occupies a singleton weak (strong) region. Vertices a, b and c are in the same natural loop, but c is in a different strong region than a and b.

4.1. Reducible control flow and loop analysis

A control flow graph is *reducible* iff for every backedge $v \rightarrow w$ (as identified by a depth-first search of the graph from *ENTRY*), either $v = w$ or w **dom** v. Each vertex w has an associated set of backedge sources

$$\text{back-srcs}(w) = \{\ v \mid v \rightarrow w \text{ is a backedge}\ \}.$$

A vertex h is a loop-entry if back-srcs(h) $\neq \emptyset$. Natural loops identify loops and loop nesting in the control flow graph [1]. The natural loop associated with loop-entry h is:

$$\text{nat-loop}(h) = \{h\} \cup \{\ v \mid \text{there is an } h\text{-free path from } v \text{ to a vertex in back-srcs}(h)\ \}$$

In reducible control flow graphs, a loop-entry h dominates every vertex in nat-loop(h) (except h itself). Roughly stated, reducibility restricts loops to have a single entry point. The exit points of nat-loop(h) are those vertices *inside* nat-loop(h) that pass control out of the loop:

$$\text{exits}(h) = \{\ v \mid \exists\ v \rightarrow w \text{ such that } v \in \text{nat-loop}(h) \text{ and } w \notin \text{nat-loop}(h)\ \}$$

If h and j are different loop-entry vertices, then either nat-loop(h) and nat-loop(j) are disjoint, or one is a subset of the other. If nat-loop(h) contains nat-loop(j) then h **dom** j. The loop-entry of the innermost loop that encloses vertex v is denoted loop-head(v). If vertex v is not in a loop then loop-head(v) = *ENTRY*.

Example. In the control flow graph in Figure 4, the edge $d \rightarrow a$ is the only backedge. Vertex a is a loop-entry with back-srcs(a) = { d }, nat-loop(a) = { a, b, c, d }, and exits(a) = { c }. Vertex a is the loop-head for vertices a, b, c and d, while *ENTRY* is the "loop-head" for vertices z and e. □

The loop information described above can be computed in $O(V+E)$ time using well-known methods, as we now outline. The main idea is to process loops from innermost to outermost, reducing a loop body to a single vertex before proceeding to process enclosing loops. First, a depth-first search computes the *back-srcs* sets. Then a post-order traversal of the depth-first search tree visits the loop-entry vertices from innermost to outermost loop (because h **dom** j implies that j will be visited before h in a post-order traversal of the depth-first search tree). Whenever a loop-entry h is encountered, the following steps are taken:

(1) Determine nat-loop(h) by traversing edges backwards, starting from vertices in (back-srcs(h)–h), until h is reached (which must occur since h dominates all vertices in nat-loop(h)), using marks to avoid visiting vertices more than once. Each edge in the subgraph induced by nat-loop(h) will be visited once. For each vertex $v \in$ nat-loop(h), loop-head(v) = h (because loops are visited from innermost to outermost). The set exits(h) can be identified during this phase as well.

(2) Transform the control flow graph by reducing the subgraph induced by nat-loop(h) to a single vertex h', eliminating all edges with endpoints inside nat-loop(h). This can be accomplished using T_1 and T_2 transformations, for example [1]. Figure 5 shows the control flow graph from Figure 4 after nat-loop(a) has been reduced.

We make the following observations about the loop analysis process:

- Let W and F be the number of vertices and the number of edges in the subgraph induced by nat-loop(h). Any operation inserted between steps (1) and (2) that runs in $O(W+F)$ time will increase the running time of the loop analysis by at most a constant factor.

- The reduction operation preserves strong regions. That is, if G is the control flow graph before nat-loop(h) is reduced and G' is the graph after the reduction, then for any pair of vertices (v,w) in G such that $v \notin$ nat-loop(h) and $w \notin$ nat-loop(h), v and w are in the same strong region in G iff v and w are in the same strong region in G'.

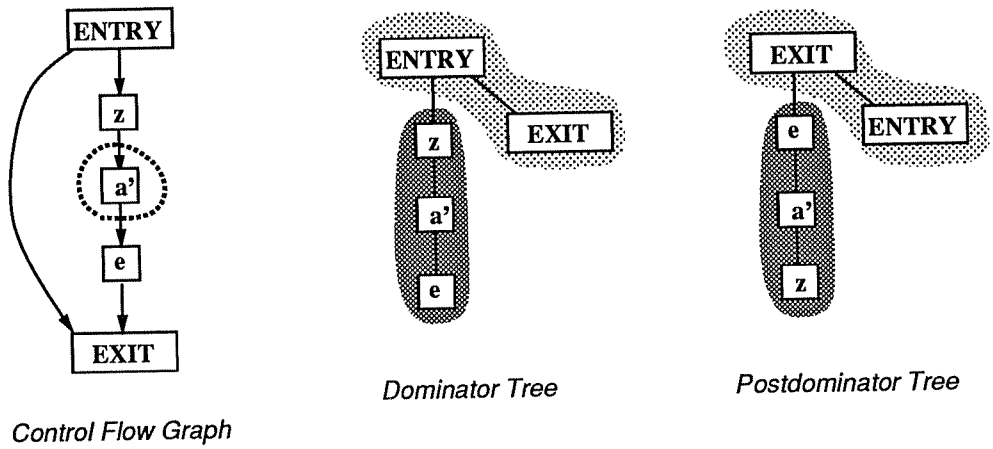


Figure 5. The control flow graph of Figure 4, after nat-loop(a) has been reduced.

4.2. Computing strong regions during loop analysis

It is clear that if loop-head(v) ≠ loop-head(w) then vertices v and w cannot be in the same strong region. For some control flow graphs, the loop-head information partitions each weak region into strong regions, as in Figure 1. However, as Figure 4 illustrates, there are control flow graphs for which this is not true. In this graph, b **dom** c, c **pd** b, and a = loop-head(b) = loop-head(c), but there is a cycle that contains vertex b and not c.

To deal with this problem it is convenient to introduce a *generalized* notion of postdominance: v **pd** w with respect to a set of vertices S iff $v \neq w$ and v is on every path from w to a vertex in S.[5] The first result of this section is that for reducible control flow graphs:

(*) Distinct vertices v and w are in the same strong region iff
(h = loop-head(v) = loop-head(w)) and ((v **dom** w and w **pd** v w.r.t. back-srcs(h) ∪ exits(h))[6]
or (w **dom** v and v **pd** w w.r.t. back-srcs(h) ∪ exits(h)))

Note the structural similarity between this definition and the definition of a weak region. The correctness of this new definition is proved at the end of the section. We first concentrate on how to use this definition to implement strong region analysis efficiently. The main idea is to identify strong regions during loop analysis, using weak region identification on each loop body (between steps (1) and (2)). The loop body is slightly transformed so that the generalized postdominance query is formed as a standard postdominance query. Let G be a reducible control flow graph and let H represent the subgraph of G induced by nat-loop(h). Graph H is transformed as follows: add a new vertex *TMP*; for each vertex $v \in$ back-srcs(h) ∪ exits(h), add an edge $v \rightarrow$*TMP*.

[5]Gupta generalized postdominance so that a set of vertices could postdominate a vertex [6]. This is different from our generalization.

[6]Note that if h = loop-head(v) = loop-head(w) and w **pd** v with respect to exits(h) then w **pd** v, since any path from a vertex in nat-loop(h) to *EXIT* must include a vertex in exits(h).

Domination and postdomination can be computed for the loop subgraph H, where vertex h acts as *ENTRY* and *TMP* acts as *EXIT*. Figure 6 illustrates the loop transformation on the control flow graph from Figure 4. Weak regions are shaded in the dominator and postdominator trees of the loop graph. Note that vertex c no longer occupies the same weak region as a and b, and that weak regions in the transformed loop graph correctly identify strong regions. The vertex *TMP* should be removed from its containing region, as it merely serves as a temporary *EXIT* vertex.

We make two observations relating dominance and postdominance in G and the loop graph H: first, because G is reducible, v **dom** w in G iff v **dom** w in H; second, w **pd** v with respect to back-srcs(h) $\cup$ exits(h) in G iff w **pd** v in H. Given the correctness of the new definition for strong region and these observations, it is fairly straightforward to see that for any pair of distinct vertices (v,w) such that h = loop-head(v) = loop-head(w), v and w are in the same strong region in G iff v and w are in the same weak region in H.

Between steps (1) and (2) of loop analysis, weak regions are identified in the (transformed) loop graph. If W and F are the number of vertices and edges in the subgraph induced by nat-loop(h), the transformed graph contains (W+1) vertices and (F + |back-srcs(h) $\cup$ exits(h)|) edges, which is clearly $O(W+F)$. Therefore, weak region analysis of the (transformed) loop graph runs in time $O(W+F)$, adding only a constant factor to the running time of the loop analysis phase.

As noted before, the reduction step (2) is guaranteed to preserve strong regions (with respect to the original vertices in the control flow graph). However, a loop may contain reduced vertices representing loops that have already been analyzed. Figure 5 shows the graph from Figure 6 after the loop has been reduced to a single vertex, a', which ends up in the weak region for vertices e and z. These reduced vertices can be

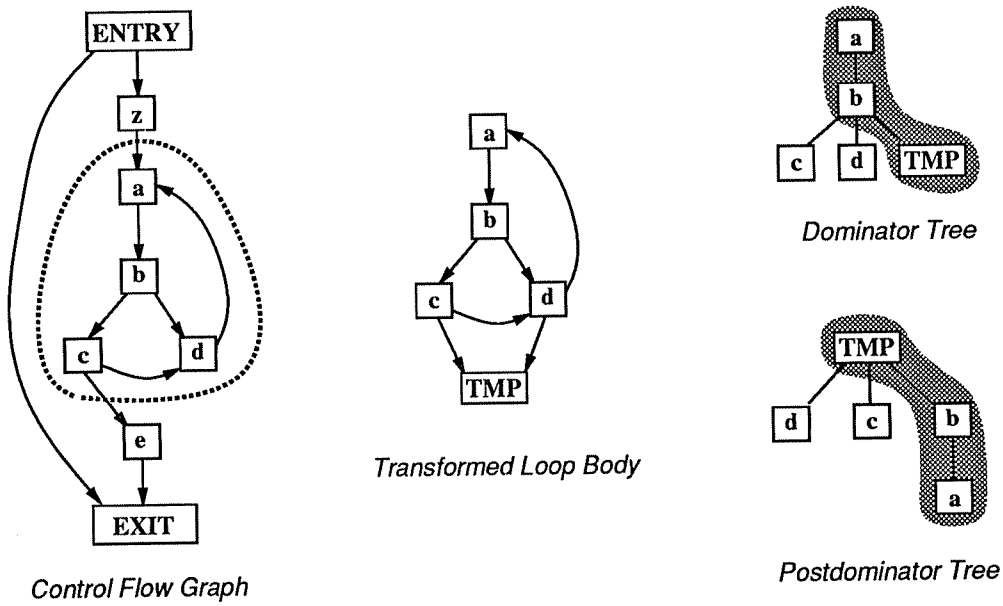


Figure 6. Weak region analysis on the (transformed) loop identifies strong regions.

eliminated from the weak regions after the regions have been identified.

We now prove two lemmas from which the main result of this section (*) follows:

LEMMA(2). Let v **dom** w and w **pd** v in a reducible control flow graph. If v is in every cycle containing w and w is in every cycle containing v, then h = loop-head(v) = loop-head(w) and w **pd** v with respect to back-srcs(h) ∪ exits(h).
PROOF. If loop-head(v) ≠ loop-head(w) then there is a cycle that contains v but not w, or vice versa. Therefore, h = loop-head(v) = loop-head(w). Since ($h = v$ or h **dom** v) and v **dom** w, there must be a w-free path from h to v. We now show that w **pd** v with respect to back-srcs(h) ∪ exits(h). If there is a w-free path from v to a vertex in back-srcs(h), then there is a cycle containing v but not w. Therefore, w **pd** v with respect to back-srcs(h).

Suppose there is a vertex $z \in$ exits(h) such that there is a w-free path from v to z. Let z' be a successor of z such that $z' \notin$ nat-loop(h). If there is a w-free path from z' to *EXIT*, then w does not postdominate v, which contradicts an initial assumption. If every path from z' to *EXIT* includes w, then the first vertex from nat-loop(h) in each such path must be h (since $z' \notin$ nat-loop(h) and $w \in$ nat-loop(h)). This implies that there is a w-free path from z to h, so there is a cycle that contains v but not w. Therefore, w **pd** v with respect to exits(h). □

LEMMA(3). For any reducible control flow graph, if h = loop-head(v) = loop-head(w), v **dom** w and w **pd** v with respect to back-srcs(h) ∪ exits(h), then: (A) v is in every cycle containing w, and (B) w is in every cycle containing v.
PROOF.

(A) Suppose there is a cycle that contains w and a backedge $y \rightarrow z$. If $z = v$ then the proof is complete. Assume that $z \neq v$. Vertex w must be a member of nat-loop(z). Since loop-head(v) = loop-head(w) and $w \in$ nat-loop(z), it follows that $v \in$ nat-loop(z) and that z **dom** v. Since z **dom** v and v **dom** w, any path from z to w must include v, so v is in the cycle.

(B) Suppose there is a cycle that contains v and a backedge $y \rightarrow z$. v must be a member of nat-loop(z), as well as nat-loop(h). Since nat-loop(h) is the innermost loop containing v, any path from v to y must contain a vertex in back-srcs(h) ∪ exits(h). Since w **pd** v with respect to back-srcs(h) ∪ exits(h), it follows that w must be in the cycle. □

4.3. Strong regions without loop analysis

A natural loop nat-loop(h) is a **while** loop if no vertex in nat-loop(h) postdominates h. A *while-graph* is a reducible control flow graph in which every natural loop is a **while** loop. The control flow graph in Figure 1 is a while-graph, but the control flow graph in Figure 4 is not. While-graphs are of interest because strong region analysis can be accomplished by weak region analysis over the entire control flow graph, followed by a simple pass over each region. No loop analysis is required for these graphs.[7] Strong regions in while-graphs can be characterized as follows:

[7]It is possible to transform any reducible control flow graph into a while-graph and preserve strong regions. To do this requires loop analysis. The transformation can be done in linear time and adds $O(V+E)$ components to the graph.

(+) In a while-graph, distinct vertices v and w are in the same strong region iff
(neither v nor w is a loop-entry) and ((v **dom** w and w **pd** v) or (w **dom** v and v **pd** w))

Thus, strong regions can be identified as follows: (1) compute weak regions; (2) for each weak region R and for each vertex v in R, if vertex v is a loop-entry (*i.e.*, is the target of a backedge), then remove v from R and put it in its own strong region. The vertices that remain in R are in the same strong region. The properties of while-graphs that lead to the simplified definition of strong regions (+) are:

(1) If h is the loop-entry of a **while** loop, then h is in a strong region by itself.
(2) In a while-graph, if v and w are distinct vertices, neither is a loop-entry, and v and w are in the same weak region then loop-head(v) = loop-head(w). This does not hold in general. For example, in Figure 4, vertices z and b are in the same weak region but not in the same loop.
(3) In a while-graph, if h = loop-head(v) = loop-head(w) then w **pd** v iff w **pd** v with respect to back-srcs(h) ∪ exits(h). Because of this property, the generalized postdominance query can be answered with a normal postdominance query.

5. CONTROL DEPENDENCE REGIONS

This section reviews the definition of control dependence and the algorithm for computing regions with control dependence. It then shows that strong regions are equivalent to control dependence regions for all control flow graphs.

In a control flow graph, vertex w postdominates the L-branch of v, denoted by w **pd** (v,L), iff w is the L-successor of v or w postdominates the L-successor of v. There is an L control dependence from vertex v to vertex w, denoted by $v \rightarrow_c^L w$, iff w **pd** (v,L) and not w **pd** v. The control dependence predecessors of w are denoted by the set CONDS(w) = { (v,L) | $v \rightarrow_c^L w$ }. Vertices v and w are in the same control dependence region iff CONDS(v) = CONDS(w). The control dependence graph contains every vertex in the control flow graph except *EXIT* and a directed edge for each control dependence $v \rightarrow_c^L w$.

Example. Figure 8(a) presents the control dependence graph of the control flow graph from Figure 1, with control dependence regions identified. These regions are equivalent to strong regions. Figure 8(b) presents the forward control dependence graph, which contains those control dependences that are not loop-carried. Regions of forward control dependence are equivalent to weak regions. □

In [4], the authors showed how regions can be computed by examining the control dependence successors of each vertex. Using the control flow graph and postdominator tree, the control dependences successors of a vertex can be enumerated in time proportional to the number of such successors [4]. Thus, the control dependence graph need not be explicitly constructed to perform region analysis (yielding an $O(V+E)$ bound on space). Unfortunately, there can be $O(V \times E)$ control dependences because each vertex can have $O(E)$ control dependence predecessors in the worst case. For example, in Figure 8, vertices b, c, and d have multiple control dependence predecessors.

We now show that for all control flow graphs, strong regions are equivalent to control dependence regions. The following two lemmas are used in the proof of this result:

LEMMA(4). Let v be a vertex ($v \neq ENTRY$ and $v \neq EXIT$) in a control flow graph. On any path *PTH* from *ENTRY* to v, there is an edge $p \rightarrow^L q$ such that $p \rightarrow_c^L v$.
PROOF. Let p be the closest vertex to the last occurrence of v in *PTH* (excluding the last occurrence of v) such that v does not postdominate p. Such a vertex must exist since no vertex except *EXIT* postdominates *ENTRY*. Let q be the successor of p in *PTH*. Let the label on edge $p \rightarrow q$ be L. Either $v = q$ or v **pd** q (otherwise there is a vertex in *PTH* closer to v that v does not postdominate). Since not v **pd** p and v **pd** (p,L),

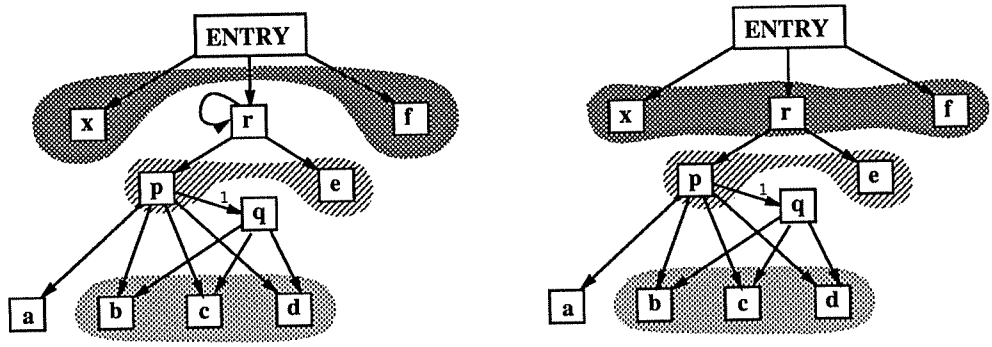


Figure 8. The full (a) and forward (b) control dependence graphs of the control flow graph in Figure 1, with regions of identical control dependence identified.

it follows that $p \longrightarrow_c^L v$. □

LEMMA(5). Let v be a vertex ($v \neq ENTRY$ and $v \neq EXIT$) in a control flow graph. On any path PTH from v to v, there is an edge $p \rightarrow^L q$ such that $p \longrightarrow_c^L v$.

PROOF. Let p be the closest vertex to the last occurrence of v in PTH (excluding the last occurrence of v) such that v does not postdominate p. Such a vertex must exist since v does not postdominate itself. If $p = v$ and v has only one control flow successor then $EXIT$ is not reachable from v, which contradicts the definition of control flow graph. Otherwise, the proof follows as in lemma (4). □

THEOREM (2). Given a control flow graph, distinct vertices v and w are in the same strong region $\Leftrightarrow$ CONDS(v) = CONDS(w).

PROOF.

($\Rightarrow$) Let p be a vertex such that $p \longrightarrow_c^L v$. We will show that $p \longrightarrow_c^L w$ must exist. A symmetric argument can be used to show that each control dependence predecessor of w is also a control dependence predecessor of v. In what follows, let $P(v)$ denote the number of occurrences of vertex v in path P.

Let P_1 be a path from $ENTRY$ to p. Since not v **pd** p there is an acyclic v-free path from one of p's successors to the $EXIT$ vertex. Let P_2 denote such a path. Let P_3 denote any acyclic path starting with the L-successor of p and ending with $EXIT$. $P_3(v) = 1$ since v **pd** (p,L) and P_3 is acyclic. Because v and w are in the same strong region, it must be the case that $P_1(v)+P_2(v) = P_1(w)+P_2(w)$ and that $P_1(v)+P_3(v) = P_1(w)+P_3(w)$. Using the facts that $P_2(v) = 0$ and $P_3(v) = 1$, these equations simplify to $P_1(v) = P_1(w)+P_2(w)$ and $P_1(v)+1 = P_1(w)+P_3(w)$. Simplifying further, we have $1 = P_3(w)-P_2(w)$. Since P_2 and P_3 are both acyclic, w can occur at most once in each path. Therefore, $P_3(w) = 1$ and $P_2(w) = 0$. These facts imply that w occurs on any path from the L-successor of p to $EXIT$ (w **pd** (p,L)) and that there is a w-free path from one of p's successors to $EXIT$ (not w **pd** p). Therefore, $p \longrightarrow_c^L w$.

($\Leftarrow$) The proof breaks into two parts:

(1) Show that every complete path that contains v also contains w and vice versa. Let PTH be a complete path that contains vertex v. By lemma (4), the prefix of PTH up to and including v must contain an edge $p \rightarrow^L q$ such that $p \longrightarrow_c^L v$. Since $p \longrightarrow_c^L w$, it follows that w **pd** (p,L). Therefore, w must

occur in *PTH*. A symmetric argument shows that v is in every complete path that contains w.

(2) Show that every cycle that contains v also contains w and vice versa. By lemma(5), any path from v to v (a cycle C) must contain an edge $p \rightarrow^L q$ such that $p \longrightarrow_c^L v$. Since $p \longrightarrow_c^L w$, it follows that w **pd** (p,L) and not w **pd** p. Suppose that cycle C does not contain w. There is a w-free path from p to p that starts with the L-branch of P (specifically, the cycle C). Since not w **pd** p, there is a w-free path from p to *EXIT*. These two facts imply that w does not postdominate the L-successor of p, a contradiction. Therefore, cycle C must contain vertex w. A symmetric argument shows that v is in every cycle that contains w. □

6. CONCLUSIONS

Regions of control dependence have a variety of uses in optimizing and parallelizing compilers, and program transformation systems. This paper has presented the first linear-time algorithm for identifying regions (without the use of control dependence) and has identified two types of regions, weak regions and strong regions. The algorithms make use of a special property of dominator and postdominator trees to compute weak regions efficiently. Combining loop analysis with weak region identification yield a linear time algorithm for computing strong regions.

ACKNOWLEDGEMENTS

Thanks to G. Ramalingam for participating in initial discussions on the topic, and to G. Ramalingam, Susan Horwitz, Jim Larus, and Samuel Bates for their comments on drafts of this paper.

REFERENCES

1. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA (1986).

2. D. Bernstein and M. Rodeh, "Global Instruction Scheduling for Superscalar Machines," *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (SIGPLAN Notices)* **26**(6) pp. 241-255 (June 1991).

3. R. Cytron, J. Ferrante, and V. Sarkar, "Experiences Using Control Dependence in PTRAN," *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, (August 1989).

4. R. Cytron, J. Ferrante, and V. Sarkar, "Compact Representations for Control Dependence," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (SIGPLAN Notices)* **25**(6)(June 20-22, 1990).

5. J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* **9**(5) pp. 319-349 (July 1987).

6. R. Gupta, "Generalized Dominators and Post-dominators," pp. 246-257 in *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, (Albuquerque, NM, January 19-22, 1992), ACM, New York (1992).

7. D. Harel, "A Linear Time Algorithm for Finding Dominators in Flow Graphs and Related Problems," *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pp. 185-194 (1985).

8. S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

PAGINATION / BACKING-UP PLANNER

FRONT	BACK	FRONT	BACK	FRONT	BACK	FRONT	BACK	FRONT	BACK
Cover	X								
1	2								
3	4								
5	6								
7	8								
9	10								
11	12								
13	14								
X	Back cover								
FRONT	BACK	FRONT	BACK	FRONT	BACK	FRONT	BACK	FRONT	BACK