What’s in a Region? or Computing control dependences in near-linear time for reducible control flow.

Thomas Ball (tball@research.bell-labs.com)

December 1993

What’s in a Region?

- OR -

Computing Control Dependence Regions in Near-linear Time for Reducible Control-flow

THOMAS BALL
University of Wisconsin—Madison

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 common algorithm for computing regions examines each control dependence at least once. As there may be $O(V \times E)$ control dependences in the worst case, where $V$ and $E$ are the number of vertices and edges in the control-flow graph, this algorithm has a worst-case running time of $O(V \times E)$. We present algorithms for finding regions in reducible control-flow graphs in near-linear time, 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.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Tools and Techniques; D.3.4 [Programming Languages]: Processors—compilers, optimizations

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Control dependence, control-flow graph, regions

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 to identify 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 partitioning of vertices that regions induce, both questions can be answered efficiently. Cytron, Ferrante and Sarkar show how regions can be computed by examining the control dependence successors of each vertex [4]. Using the control-flow graph and postdominator tree, the control...
dependence successors of a vertex can be enumerated in time proportional to the number of such successors. Unfortunately, there can be $O(V \times E)$ control dependences in the worst-case (even for acyclic control-flow graphs). Their algorithm uses $O(V + E)$ space.

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

- 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 control dependences [10].

- 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. As we show, 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 $O(V + E \alpha(E, V))$ time (where $\alpha(E, V)$ is related to the inverse of Ackermann’s function and $\alpha(E, V) \leq 3$ nearly always) and $O(V + E)$ space for reducible control-flow graphs. Weak regions can 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. 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.

This paper makes two contributions:

(1) 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.

\(^1\) A complete control-flow path is a path from the entry point of the control-flow graph to the exit point.
A nearly-linear algorithm for computing regions of control dependence. Algorithms that examine control dependences have poor worst-case performance because there are situations in which $O(V \times E)$ control dependences arise (although these may be infrequent in practice). Our algorithm performs well in all cases, as its time complexity is not dependent on the number of control dependences.

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. Section 6 reviews related work and Section 7 concludes.

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.

2.2. Applications of regions

The Introduction presented the definitions of weak and strong regions. Figure 1 presents a 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. 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 \( ENTRY \rightarrow x \rightarrow r \rightarrow f \rightarrow EXIT \). 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 \( v \text{ dom} w \), if \( v \neq w \) and \( v \) is on every path from \( ENTRY \) to \( w \).\(^2\) Vertex \( v \) immediately dominates \( w \), denoted by \( v \text{ idom} w \), if \( v \text{ dom} w \) and there is no vertex \( z \) such that \( v \text{ dom} z \text{ dom} w \). Vertex \( v \) postdominates vertex \( w \), denoted by \( v \text{ pd} w \), if \( v \neq w \) and \( v \) is on every path from \( w \) to \( EXIT \). Immediate postdominance \( (v \text{ ipd} w) \) 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.

As defined, the dominator (postdominator) relation is an irreflexive, asymmetric, transitive relation. The dominator (postdominator) relation can be represented as a tree where \( v \) is the parent of \( w \) iff \( v \text{ idom} w \) \((v \text{ ipd} w)\) and \( v \) is a proper ancestor of \( w \) iff \( v \text{ dom} w \) \((v \text{ pd} w)\). Figure 2 presents the dominator and postdominator trees for the control-flow graph from Figure 1. 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 easy to see that distinct vertices \( v \) and \( w \)

\(^2\)This differs slightly from the usual definition of dominance, which is reflexive.
are in the same weak region iff (v $\text{dom}$ w and w $\text{pd}$ v) or (w $\text{dom}$ v and v $\text{pd}$ w). In Figure 1, p $\text{dom}$ e and e $\text{pd}$ p, so vertices p and e are 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 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 $\text{dom}$ c and c $\text{pd}$ a then a $\text{dom}$ b $\text{dom}$ c iff c $\text{pd}$ b $\text{pd}$ a.

**Proof.**

($\Rightarrow$) Suppose that a $\text{dom}$ b $\text{dom}$ c. This implies that every path from a to c includes b. Since c $\text{pd}$ a, every path from a to EXIT includes c, so every path from a to EXIT must also include b (b $\text{pd}$ a). Suppose that there is a c-free path from b to EXIT (not c $\text{pd}$ b). Since a $\text{dom}$ b $\text{dom}$ c, 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 $\text{pd}$ b.

($\Leftarrow$) Because postdominators can be defined as dominators in the reverse control-flow graph, the proof is similar 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 $\text{dom}$ w and w $\text{pd}$ v) or (w $\text{dom}$ v and v $\text{pd}$ w). As 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 $v_1$ $\text{pd}$ $v_n$, it follows that $v_n$ $\text{dom}$ $v_1$. Consider any vertex $v_i$, where $1 < i < n$. Since $v_1$ $\text{pd}$ $v_i$ $\text{pd}$ $v_n$ and $v_n$ $\text{dom}$ $v_1$,

---

3This definition is identical to Bernstein and Rodeh’s notion of equivalent vertices [2]. However, they use control dependences to discover the classes of equivalent vertices, which is not as efficient as the method given here.
lemma (1) implies that $v_n \text{ dom } v_i$, so $v_i$ must be in $R$.

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

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

- $\text{WREGION}(v)$: the weak region number associated with vertex $v$.
- $\text{WHEAD}(R)$: the first vertex in weak region $R$ (i.e., lowest in postdominator tree).
- $\text{WTAIL}(R)$: the last vertex in weak region $R$ (i.e., highest in postdominator tree).
- $\text{WNEXT}(v)$: the vertex after $v$ in $\text{WREGION}(v)$ (i.e., $\text{WNEXT}(v) \text{ ipd } v$).
- $\text{WPREV}(v)$: the vertex before $v$ in $\text{WREGION}(v)$ (i.e., $\text{v ipd WPREV(v)}$).

For each vertex in the control-flow graph, three pieces of information are maintained ($\text{WREGION}$, $\text{WNEXT}$, and $\text{WPREV}$), and for each weak region, two pieces of information are needed ($\text{WHEAD}$ and $\text{WTAIL}$). As 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 $\text{region_num}$ keeps track of the number of weak regions (chains) found so far. The depth-first search is done on the postdominator tree (although it could just as easily be done on the dominator tree). $\text{EXIT}$ 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

$\text{A chain in a tree } T \text{ is a sequence of vertices } (v_1, \cdots, v_n) \text{ such that for all } i, v_j \text{ is a parent of } v_{i+1} \text{ in } T.$
equivalent. However, for cyclic control-flow graphs, two vertices may be in the same weak region but in different strong regions. 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 the same 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. 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 \( b \). It is straightforward to see that an equivalent characterization of strong regions is:

\[
\text{Distinct vertices } v \text{ and } w \text{ are in the same strong region iff }\\
(v \ \text{dom}\ w \text{ and } w \ \text{pd}\ v) \text{ or } (w \ \text{dom}\ v \text{ and } v \ \text{pd}\ w) \text{ and }\\(v \text{ is in every cycle containing } w) \text{ and } (w \text{ is in every cycle containing } v)\\
\]

This section describes how to compute strong regions efficiently for reducible control-flow graphs, using loop analysis in conjunction with weak region identification. We first review the concepts of reducibility and natural loop analysis and then show how to compute strong regions efficiently.

### 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 \( \text{ENTRY} \)), either \( v = w \) or \( w \ \text{dom}\ v \). 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 \( \text{back-srcs}(h) \neq \emptyset \). Natural loops identify loops and loop nesting in the control-flow graph [1]. The natural loop associated with loop-entry \( h \) is:

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

The exit points of \( \text{nat-loop}(h) \) are those vertices in \( \text{nat-loop}(h) \) that pass control out of the loop:

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

In a reducible control-flow graph, a loop-entry \( h \) dominates every vertex in \( \text{nat-loop}(h) \) (except \( h \) itself). If \( h \) and \( j \) are different loop-entry vertices, then either \( \text{nat-loop}(h) \) and \( \text{nat-loop}(j) \) are disjoint, or one is a
subset of the other. The loop-entry of the innermost loop that encloses vertex \( v \) is denoted by \( \text{loop-head}(v) \). If vertex \( v \) is not in a loop then \( \text{loop-head}(v) = \text{ENTRY} \).

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

The above loop information can be computed in \( O(V + E \alpha(E, V)) \) time using well-known methods [11, 12], where \( \alpha(E, V) \) is related to the inverse of Ackermann’s function and is nearly always less than three. The main idea is to process loops from innermost to outermost, reducing a loop body to a single vertex before processing enclosing loops. A depth-first search computes the \( \text{back-srcs} \) sets, followed by a post-order traversal of the depth-first search tree that visits the loop-entry vertices from innermost to outermost loop (because \( h \; \text{dom} \; j \) implies that \( j \) will be visited before \( h \) in a post-order traversal). Whenever a loop-entry \( h \) is encountered, the following steps are taken:

1. Determine \( \text{nat-loop}(h) \) by traversing edges backwards, starting from vertices in \( (\text{back-srcs}(h) - h) \), until \( h \) is reached, using marks to avoid visiting vertices more than once. For each vertex \( v \in \text{nat-loop}(h) \), \( \text{loop-head}(v) = h \). Any vertex \( v \) in \( \text{nat-loop}(h) \) with an outgoing edge not touched by the traversal is in \( \text{exits}(h) \).

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

We make the following observations about the loop analysis process (which will be made use of later):

- Let \( W \) and \( F \) be the number of vertices and the number of edges in the subgraph induced by \( \text{nat-loop}(h) \). 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 \( v \not\in \text{nat-loop}(h) \) and \( w \not\in \text{nat-loop}(h) \), \( v \) and \( w \) are in the same strong region in \( G \) iff \( v \) and \( w \) are in the same strong region in \( G' \).

4.2. Computing strong regions during loop analysis

We introduce a generalized notion of postdominance in order to compute strong regions: \( v \text{ pd } w \) with respect to a set of vertices \( S \) iff \( v \neq w \) and \( v \) is on every path from \( w \) to a vertex in \( S \). The first result of this section is that for reducible control-flow graphs:

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

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 \( \text{TMP} \); for each vertex \( v \in \text{back-srcs}(h) \cup \text{exits}(h) \), add an edge \( v \rightarrow \text{TMP} \).

Domination and postdomination can be computed for the loop subgraph \( H \), where vertex \( h \) acts as \( \text{ENTRY} \) and \( \text{TMP} \) acts as \( \text{EXIT} \). 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 \( \text{TMP} \) should be removed from its containing region, as

---

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

6Note that if \( h = \text{loop-head}(v) = \text{loop-head}(w) \) and \( w \text{ pd } v \) with respect to \( \text{exits}(h) \) then \( w \text{ pd } v \), since any path from a vertex in nat-loop\( (h) \) to \( \text{EXIT} \) must include a vertex in \( \text{exits}(h) \).
it merely serves as a temporary EXIT vertex.

Two observations relating dominance and postdominance in $G$ and the loop graph $H$ are of importance:

1. Because $G$ is reducible, $v \text{ dom } w$ in $G$ iff $v \text{ dom } w$ in $H$; 2. $w \text{ pd } v$ with respect to back-srcs$(h) \cup$ exits$(h)$ in $G$ iff $w \text{ pd } v$ in $H$. Given the correctness of the new definition for strong region and these observations, it is straightforward to see that for any pair of distinct vertices $(v, w)$ such that $h = \text{loop-head}(v) = \text{loop-head}(w)$, $v$ 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 $(F + |\text{back-srcs}(h) \cup \text{exits}(h)|)$ edges, which is clearly $O(W + F)$. 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 weak region may contain reduced vertices representing loops that have already been analyzed (e.g., vertex $a'$ in Figure 5). These vertices can be 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 $v \text{ dom } w$ and $w \text{ pd } v$ in a reducible control-flow graph. If $v$ is in every cycle containing $w$ and $w$ is in every cycle containing $v$, then $h = \text{loop-head}(v) = \text{loop-head}(w)$ and $w \text{ pd } v$ with respect to back-srcs$(h) \cup$ exits$(h)$.

**Proof.** If $\text{loop-head}(v) \neq \text{loop-head}(w)$ then there is a cycle that contains $v$ but not $w$, or vice versa, so $h = \text{loop-head}(v) = \text{loop-head}(w)$. Since $(h = v$ or $h \text{ dom } v)$ and $v \text{ dom } w$, there must be a $w$-free path from $h$ to $v$. We now show that $w \text{ pd } v$ with respect to back-srcs$(h) \cup$ 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, $w \text{ pd } v$ with respect to back-srcs$(h)$.

Suppose there is a vertex $z \in \text{exits}(h)$ such that there is a $w$-free path from $v$ to $z$. Let $z'$ be a successor of $z$ such that $z' \notin \text{nat-loop}(h)$. 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 $z' \notin \text{nat-loop}(h)$ and $w \in \text{nat-loop}(h)$). 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, \( w \text{ pd } v \) with respect to \( \text{exits}(h) \). \( \square \)

**Lemma (3).** For any reducible control-flow graph, if \( h = \text{loop-head}(v) = \text{loop-head}(w) \), \( v \text{ dom } w \) and \( w \text{ pd } v \) with respect to \( \text{back-srcs}(h) \cup \text{exits}(h) \), 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 \( \text{nat-loop}(z) \). Since \( \text{loop-head}(v) = \text{loop-head}(w) \) and \( w \in \text{nat-loop}(z) \), it follows that \( v \in \text{nat-loop}(z) \) and that \( z \text{ dom } v \). Since \( z \text{ dom } v \) and \( v \text{ dom } w \), 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 \( \text{nat-loop}(z) \), as well as \( \text{nat-loop}(h) \). Since \( \text{nat-loop}(h) \) is the innermost loop containing \( v \), any path from \( v \) to \( y \) must contain a vertex in \( \text{back-srcs}(h) \cup \text{exits}(h) \). As \( w \text{ pd } v \) with respect to \( \text{back-srcs}(h) \cup \text{exits}(h) \), it follows that \( w \) must be in the cycle. \( \square \)

### 5. Control Dependence Regions

This section reviews the definitions of control dependence and region of 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 \( w \text{ pd } (v, L) \), iff \( w \) is the \( L \)-successor of \( v \) (i.e., the edge \( v \rightarrow w \) exists) or \( w \) postdominates the \( L \)-successor of \( v \). There is an \( L \) control dependence from vertex \( v \) to vertex \( w \), denoted by \( v \rightarrow w \), iff \( w \text{ pd } (v, L) \) and not \( w \text{ pd } v \). The control dependence predecessors of \( w \) are denoted by the set \( \text{CONDS}(w) = \{ (v, L) \mid v \rightarrow w \} \). Vertices \( v \) and \( w \) are in the same control dependence region iff \( \text{CONDS}(v) = \text{CONDS}(w) \). The control dependence graph contains every vertex in the control-flow graph except \( \text{EXIT} \) and a directed edge for each control dependence \( v \rightarrow w \).

Figure 7(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 7(b) presents the forward control dependence graph. Regions of forward control dependence are equivalent to weak regions.
Podgurski has shown that regions of forward control dependence are equivalent to weak regions for all control-flow graphs [10]. We show that for all control-flow graphs, strong regions are equivalent to regions of full control dependence. The following two lemmas are used in the proof of this result:

**LEMMA (4).** Let $v$ be a vertex ($v \neq \text{ENTRY}$ and $v \neq \text{EXIT}$) in a control-flow graph. On any path $PTH$ from $\text{ENTRY}$ to $v$, there is an edge $p \rightarrow L q$ such that $p \rightarrow L v$.

**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 $\text{EXIT}$ is not reachable from $v$, which contradicts the definition of control-flow graph. Otherwise, the proof follows as in lemma (4). □

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

**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 $\text{EXIT}$ 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 iff $\text{CONDS}(v) = \text{CONDS}(w)$.

**PROOF.**

$(\Rightarrow)$ Let $p$ be a vertex such that $p \rightarrow L v$. We will show that $p \rightarrow L w$ 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 $\text{ENTRY}$ to $p$. Since not $v$ pd $p$ there is an acyclic $v$-free path from one of $p$’s successors to the $\text{EXIT}$ 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 $\text{EXIT}$. $P_3(v) = 1$ 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, so
$P_3(w) = 1$ and $P_2(w) = 0$. These facts imply that $w$ occurs on any path from the $L$-successor of $p$ to $\text{EXIT}(w \ \text{pd} \ (p, L))$ and that there is a $w$-free path from one of $p$'s successors to $\text{EXIT}$ (not $w \ \text{pd} \ p$). Therefore, $p \rightarrow^L_c w$.

(\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 $p \rightarrow^L c v$. Since $p \rightarrow^L c w$, it follows that $w \ \text{pd} \ (p, L)$, so $w$ 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 $p \rightarrow^L c v$. Since $p \rightarrow^L c w$, it follows that $w \ \text{pd} \ (p, L)$ and not $w \ \text{pd} \ p$. 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 $w \ \text{pd} \ p$, there is a $w$-free path from $p$ to $\text{EXIT}$. 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 $w$. \hfill \Box

6. RELATED WORK

Podgurski [10] has shown that the common algorithm for computing forward control dependences (by which forward control dependences have been defined) [3] is problematic and presents a new definition of forward control dependence. He proves that regions of forward control dependence are equivalent to what we call weak regions and develops an algorithm for computing weak regions using dominators and post-dominators that is essentially identical to ours. Our work differs in two main aspects. First, we use a different conceptual approach for defining regions of control dependence based on execution frequency (rather than in terms of control dependence itself). Second, while Podgurski focuses only on regions arising from forward control dependence, we show how to compute strong regions (regions arising from full control dependence).

Since our algorithm was originally developed, Johnson, Pearson, and Pingali have developed an algorithm for finding strong regions for which they claim linear-time performance on arbitrary control-flow graphs [9]. They use a characterization of strong regions that is nearly identical to ours. However, their algorithm does not require construction of either the dominator or postdominator relation.
7. CONCLUSIONS

Regions of control dependence have a variety of uses in optimizing and parallelizing compilers, and program transformation systems. This paper has presented nearly-linear algorithms for identifying regions in reducible control-flow graphs (without the use of control dependence), and identified two types of regions, weak regions and strong regions. Weak regions are computed using the dominator and postdominator trees and strong regions are computed through a combination of loop analysis and weak region identification.

ACKNOWLEDGEMENTS

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


Figure 1. A control-flow graph with weak and strong regions identified.
**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.
begin
[1] compute postdominator and dominator trees;
[2] region_num := 1;
[4] WNEXT(EXIT) := nil;
[5] DFS(EXIT, 1);
end

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
[12] region_num := region_num + 1;
[13] WTAIL(region_num), WNEXT(w) := w, nil;
[14] DFS(w, region_num);
fi
do
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.
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 $b$. 
Figure 5. The control-flow graph of Figure 4, after nat-loop(a) has been reduced.
Figure 6. Weak region analysis on the (transformed) loop identifies strong regions.
Figure 7. The full (a) and forward (b) control dependence graphs of the control-flow graph in Figure 1, with regions of identical control dependence identified.