

Compact Representations for Control Dependence

Ron Cytron*
Jeanne Ferrante*
Vivek Sarkar*

1 Introduction

Recently the *Program Dependence Graph* (PDG) has been shown useful as a basis for solving a variety of problems, including optimization [FOW87], vectorization [BB89], translation to dataflow machines [OBM90], code generation for VLIW machines [GS87a, GS87b], program transformation [Sel89, CF89], merging versions of programs [HPR87], and automatic detection and management of parallelism [ABC*87, ABC*88, CFS89]. The edges of the PDG consist of *control dependence* and *data dependence* edges. The data dependence edges represent the essential data flow relationships of a program [Kuc78]. In this paper, we examine the control dependence aspect of the PDG, which summarizes essential control flow relationships in a program. Informally, for nodes X and Y in CFG , Y is control dependent on X if during execution, X can directly affect whether Y is executed. We improve the space and time required to compute those aspects of control dependence used by most algorithms.

We assume a control flow graph CFG is augmented with an entry node ($START$), an exit node ($STOP$), and an edge from $START$ to $STOP$. An example control flow graph is shown in Figure 1 and its control dependence graph is given in Figure 2. Let N and E be the number of nodes and edges respectively in CFG . Multiple outgoing edges from a node in CFG are assumed to be distinctly labelled.¹ We further assume that for any node X in CFG there exists a path from $START$ to X and a path from X to $STOP$.

To define control dependence formally, we first recall the notion of *postdominance* [FOW87]. Let X and Y be nodes in CFG . If $X \neq Y$ appears on every path from Y to $STOP$, then X *postdominates* Y , denoted $X \gg Y$. We write $X \geq Y$ if $X = Y$ or $X \gg Y$. The *immediate postdominator* of Y is the closest postdominator of Y on any path from Y to $STOP$.

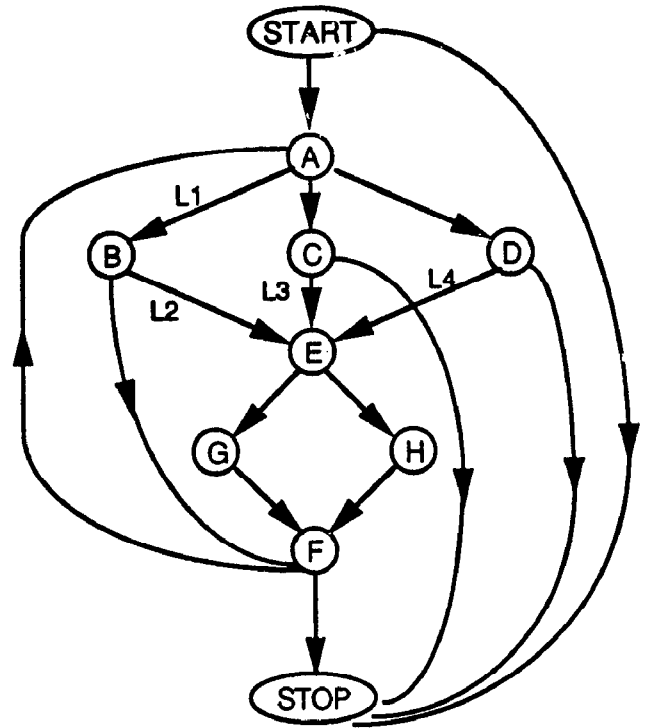


Figure 1. Control flow graph

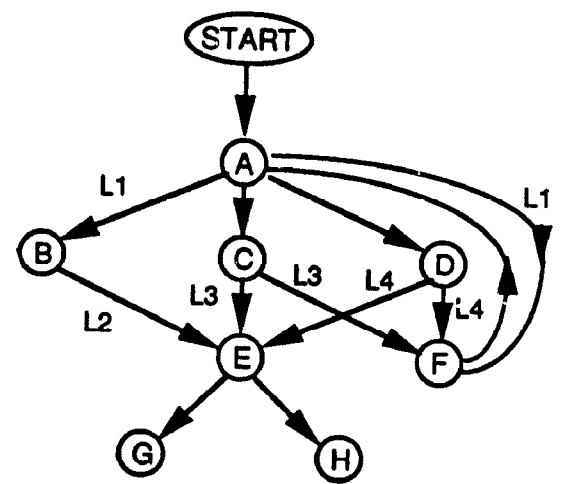


Figure 2. Control dependence graph

*IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.

¹To avoid clutter, Figure 1 shows only those four labels relevant to our discussion.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Definition 1 Let X and Y be nodes in CFG. There is a control dependence from X to Y with label L if and only if

1. There is a nonnull path $p : X \xrightarrow{L} Y$, starting with the edge labelled L , such that Y postdominates every node strictly between X and Y on p .
2. The node Y does not postdominate the node X .

To show that this definition corresponds to our informal notion of control dependence, first suppose condition 1 fails. Then either there is no such path from X to Y , or on any such paths p , there is some node W between X and Y not postdominated by Y . In the former case, X cannot affect Y 's execution; in the latter case, W is a *closer* node than X that can affect the execution of Y . If condition 2 fails, so Y postdominates X , Y would execute whenever X does, and hence X would not affect Y 's execution.

Algorithms that use control dependence typically ask one or more of the following questions:

1. What nodes are control dependent on X ?
2. What nodes have the same control dependences as node Y ?
3. On what nodes is Y control dependent?

Based on Definition 1, we can build a *control dependence graph*, CDG , containing exactly the edges of the form (X, Y, L) that satisfy the above conditions. Also, *region* nodes can be inserted to summarize nodes with identical control dependences [FOW87]. Of course, each of the above questions can be answered by consulting CDG (or its inverse). Unfortunately, the size of the control dependence graph can be quadratic in the size of the original program. Such behavior can occur in structured programs that contain `repeat...until` constructs [CFR*89], and in unstructured programs without any loops. While this worst case behavior is not expected in most cases, the possibility of its occurrence is problematic for systems like PTRAN [ABC*87] that use control dependence. Should quadratic space be dedicated for control dependence, just to accommodate exceptional cases? This seems unreasonable, but any lower fixed allocation will fail on such programs. Even an adaptive data structure potentially uses quadratic space on some programs.

Because algorithms that use control dependence do not typically require storage of the *entire* relation to answer the above questions, it suffices to compute the following sets:

$$CD(X, L) = \{ Y \mid Y \text{ is control dependent on } X \text{ with label } L \}$$

$$IDENTCD(Y) = \{ Z \mid CONDS(Y) = CONDS(Z) \}$$

$$COND(X, L) = \{ (X, L) \mid Y \in CD(X, L) \}$$

In this paper, we present efficient algorithms that use only *linear* space to compute the first two structures. Many uses of control dependence do not require $COND$ s. For example, the first two structures suffice for the identification and generation of concurrently executed statements [CFS89]. However, recent work showing the relationship between control dependence and Static Single Assignment (SSA) form essentially requires computing $COND$ s. For the $COND$ s structure, we improve both the time bound by a factor of N and the space bound in practice

by a constant factor. Unfortunately, computing $COND$ s seems to require more space than the other structures. We comment on the difficulties of maintaining a linear space bound for $COND$ s in Section 7. Also, we show in Section 6 that $COND$ s requires only linear space in practice.

We present the following (where N and E are the number of nodes and edges in a given control flow graph for a program):

- A sparse representation of control dependence for computing $CD(X, L)$ using $O(N + E)$ space. In this representation, the control dependence graph successors of a node can be enumerated in time proportional to the number of successors.
- A new algorithm for determining *regions* of control dependence which is a factor of $O(N)$ more efficient in time and space than the algorithm of [FOW87]. $IDENTCD(Y, Z)$ is easily computed, because identically control dependent nodes belong to the same region. $CD(X, L)$ is still easily computed after regions have been identified.
- Two new *factored* control dependence graph representations that improve the worst case bounds of [FOW87] by a factor of N in time and a constant factor in space. Both representations allow $COND(Y)$ to be computed in time proportional to the size of the resulting set. Also, $CD(X, L)$ and $IDENTCD(Y, Z)$ are still easily computed.
- Statistics for each of the above computations that show the space saved by our methods.

We begin by describing the worst-case quadratic behavior of the control dependence graph in Section 2. In Section 3, we show how to compute $CD(X, L)$ using linear space and time. In Section 4, we first show how regions of control dependence can be efficiently identified to compute $IDENTCD(Y, Z)$. We introduce two new methods for computing a factored control dependence graph in Section 5. We present data that shows the space saved by our methods in Section 6. In Section 7, we discuss open problems and future work.

2 Size of the Control Dependence Graph

Unfortunately, the size of the control dependence graph can be quadratic in the size of the control flow graph. This worst case can be achieved even in structured programs, such as a nest of `repeat...until` loops [CFR*89]. Here, in every loop, all nodes in the loop are control dependent on the exit tests of all surrounding loops. For N nested loops, this yields a control dependence relation of size $O(N^2)$. It is useful to note that in this case, a quadratic representation can be avoided by augmenting the control flow graph with interval entry and exit nodes for each loop [CFS89].

This quadratic size can also occur in programs without loops. In Figure 3, we show a control flow graph with $O(N)$ nodes and edges, arising from a program consisting of N *ifs* with *go to*'s. Here, the size of the control dependence relation is $O(N^2)$, since each node X_i is control

dependent on each element of the set $\{Y_1 \dots Y_i\}$. Here, and elsewhere in the paper, we omit labels associated with control dependences when they are clear from context.

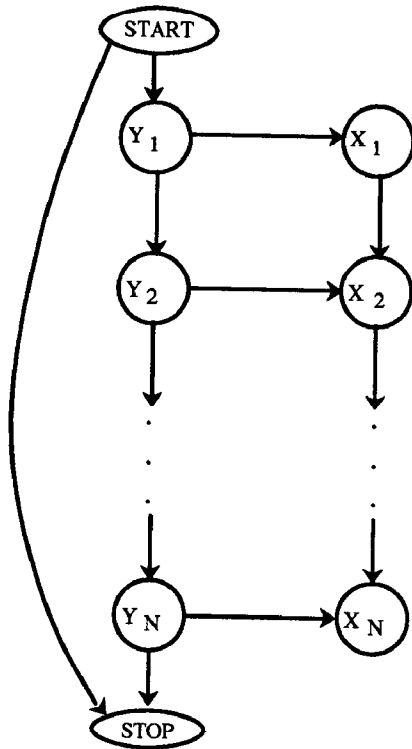


Figure 3. Quadratic space control dependence relation

The example in Figure 3 can be stored in linear space using [FOW87] and our compact representations. In Figures 7 and 10, we give examples where our methods use linear space but [FOW87] uses quadratic space. Such behavior cannot occur for programs with certain structure [CFR*89]. As we show in Section 6, such behavior is not expected in programs of arbitrary structure. However, the possibility of quadratic space cannot be ignored by systems that require control dependence information for programs of arbitrary structure. In the following sections, we show how to represent efficiently those aspects of control dependence used by most algorithms.

3 Constructing Control Dependences in Linear Time and Space

In this section, we restate a portion of the control dependence algorithm [FOW87]. Instead of computing the full control dependence graph, we retain two structures, $CD.START(X, L)$ and $CD.END(X, L)$, that allow efficient computation of $CD(X, L)$:

1. Compute $PDOM$, the postdominator tree² [FOW87], represented by:

²In a postdominator tree, the children of a node X are all immediately postdominated by X .

- (a) The array, $PDOM.PARENT(1..N)$, where $PDOM.PARENT(X)$ is node X 's immediate postdominator.
- (b) $PDOM.CHILDREN(1..N)$, the inverse of $PDOM.PARENT$ stored as an array of lists. $PDOM.CHILDREN(X)$ is a list of nodes that are immediately postdominated by node X .

Since $PDOM$ is a tree, it occupies only $O(N)$ space. $PDOM$ can be constructed in $O(N + E)$ time [Har85]. Figure 4 shows the postdominator tree for the control flow graph of Figure 1.

2. Define $CD.START(X, L) = Z$ if and only if there is an edge from X to Z labelled L in CFG . $CD.START(X, L)$ represents the start of the list of nodes control dependent on X with label L . No extra storage or time is required for $CD.START$ since the information is already available in CFG . We define the name $CD.START$ for convenience.
3. Define $CD.END(X, L) = PDOM.PARENT(X)$. $CD.END(X, L)$ represents the (non-inclusive) end of the list of nodes control dependent on X with label L . No extra storage or time is required for $CD.END$ since the information is already available in $PDOM$. We define the name $CD.END$ for convenience. Note that the value of $CD.END(X, L)$ depends only on X , and not on L .

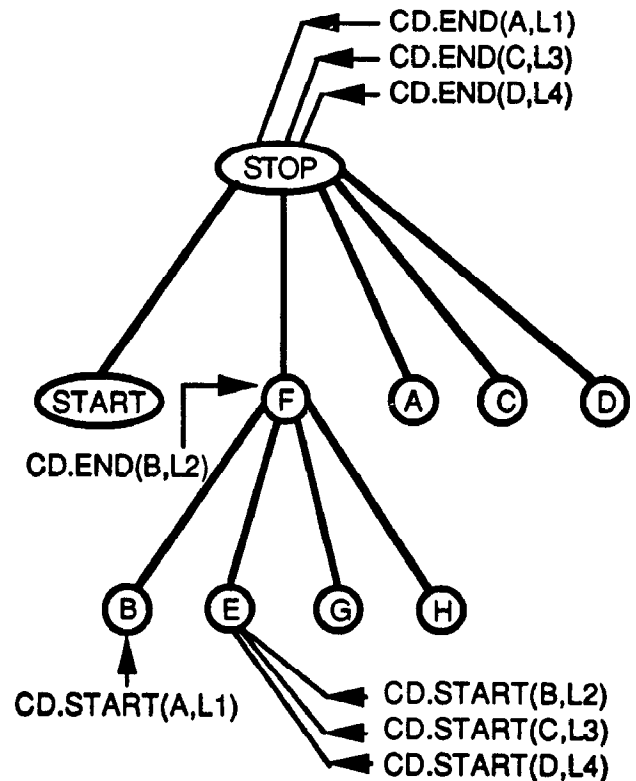


Figure 4. Postdominator tree

In Figure 4, $CD.START$ and $CD.END$ are shown annotating $PDOM$. The set $CD(X, L)$ is computed by starting at $CD.START(X, L)$ and following $PDOM.PARENT$ pointers until $CD.END(X, L)$ is

reached, as shown in the algorithm of Figure 5. Note

1. $CD(X, L) = \emptyset$
2. $Y = CD.START(X, L)$
3. while $Y \neq CD.END(X, L)$ do
 - (a) $CD(X, L) = CD(X, L) \cup \{Y\}$
 - (b) $Y = PDOM.PARENT(Y)$

Figure 5. Computation of $CD(X, L)$.

that this algorithm never adds $CD.END(X, L)$ to the set $CD(X, L)$; if $CD.START(X, L) = CD.END(X, L)$ then the algorithm computes $CD(X, L) = \emptyset$, which means that there are no nodes control dependent on X with label L . For convenience, we sometimes represent $CD(X, L)$ as a half-closed, half-open interval, $[CD.START(X, L), CD.END(X, L))$.

In [FOW87], the above algorithm is essentially applied to each node X and label L from X in the CFG . All resulting sets $CD(X, L)$ are copied to a separate data structure to form the control dependence graph [FOW87], which may take quadratic space. We use pointers into $PDOM$ (namely $CD.START$ and $CD.END$) to store compactly the information necessary for computing $CD(X, L)$. Since the size of each set $CD(X, L)$ is limited by the depth of the postdominator tree, $CD(X, L)$ can be computed using linear space.

Before proving that this algorithm correctly computes control dependence, we show that the interval

$$[CD.START(X, L), CD.END(X, L))$$

is well-defined.

Lemma 1

$$CD.END(X, L) \geq CD.START(X, L)$$

Proof. (by contradiction, based on a similar result in [FOW87].) If $CD.END(X, L) \neq CD.START(X, L)$ then assume that $CD.END(X, L)$, which is computed as $PDOM.PARENT(X)$, does not postdominate $CD.START(X, L)$. Then there would be a path in CFG from $CD.START(X, L)$ to $STOP$ that does not pass through $CD.END(X, L)$. Prefixing this path by the edge from X to $CD.START(X, L)$ gives a path from X to $STOP$ that does not pass through $CD.END(X, L)$, which implies that $CD.END(X, L) = PDOM.PARENT(X)$ does not postdominate X , a contradiction. \square

Theorem 1 *The above algorithm correctly computes the control dependence relation.*

Proof. We will show that the conditions in Definition 1 are satisfied if and only if Y occurs in the interval

$$[CD.START(X, L), CD.END(X, L))$$

1. *IF:* It is easy to see that conditions 1 and 2 in Definition 1 are satisfied if Y occurs in the interval. The

required nonnull path p for condition 1 would just be

$$\begin{aligned} X &\rightarrow CD.START(X, L) \\ &\rightarrow PDOM.PARENT(CD.START(X, L)) \\ &\rightarrow \dots \rightarrow Y \end{aligned}$$

For condition 2, we see that Y cannot postdominate X , since $CD.END(X, L) = PDOM.PARENT(X)$ postdominates Y .

2. *ONLY IF:* Now, we assume that conditions 1 and 2 in Definition 1 are known to be true, and prove that Y must occur in $[CD.START(X, L), CD.END(X, L))$. From condition 1, we see that the path p must be of the form

$$\begin{aligned} X &\rightarrow CD.START(X, L) \\ &\rightarrow PDOM.PARENT(CD.START(X, L)) \\ &\rightarrow \dots \rightarrow Y \end{aligned}$$

since $CD.START(X, L)$ is node X 's successor in CFG with label L . Since p is a non-null path, either $Y = CD.START(X, L)$ or Y postdominates $CD.START(X, L)$. Therefore, we must eventually reach Y by starting at $CD.START(X, L)$, and following $PDOM.PARENT$ pointers up the postdominator tree.

The only question that remains is whether we will reach Y before we reach $CD.END(X, L)$. This is shown to be true by condition 2, which states that Y does not postdominate X . If we reached Y at or after $CD.END(X, L) = PDOM.PARENT(X)$, it would imply that Y postdominates X , thus contradicting condition 2. \square

We have shown how to compute $CD(X, L)$ (i.e., control dependence successors of X with label L) using time proportional to the size of the set $CD(X, L)$, as in previous work [FOW87]. However, we have reduced the required space to $O(N + E)$ using $CD.START(X, L)$ and $CD.END(X, L)$. Unfortunately, this representation does not allow easy computation of $COND(X)$ (i.e., the control dependence predecessors of X) or $IDENTCD(X)$. In the following sections we consider the efficient computation of these structures.

4 Constructing Regions in Linear Space

We now consider how to compute $IDENTCD(X)$ by identifying regions [FOW87] of the control dependence graph. Two nodes belong to the same region if and only if they have the same set of control conditions in the control dependence relation. Regions therefore partition the set of CFG nodes into equivalence classes; $IDENTCD(X)$ is the set of all nodes in the equivalence class with X .

Previous work [FOW87] identifies regions by inserting region nodes in the control dependence graph in expected time $O(N)$, worst-case time $O(N^2 \times E)$, and space $O(N \times E)$. The expected time result assumed $O(1)$ time to check (by hashing) if a newly computed set of control conditions is the same as any previously computed set. However, this

operation may take $O(N \times E)$ time in the worst case, since there may be $O(N)$ previously computed sets, and each set may have $O(E)$ control conditions. In this section, we present a new algorithm that computes regions using $O(N \times E)$ time (an improvement by an $O(N)$ factor over the algorithm in [FOW87]) and $O(N + E)$ space.

- Initialization as described in text
- $NewRegion(0) = NumRegions = 0$
- for each (X, L) (edge of CFG)
 1. $T = NumRegions$
 2. $Y = CD.START(X, L)$
 3. while $Y \neq CD.END(X, L)$ do
 - (a) $R = REGION(Y)$
 - (b) if not

$$RHEAD(R) \succeq CD.START(X, L)$$

and

$$CD.END(X, L) \succcurlyeq RTAIL(R)$$

then

- i. if $NewRegion(R) \leq T$ then
 - $NumRegions = NumRegions + 1$
 - $RHEAD(NumRegions) = \perp$
 - $RTAIL(NumRegions) = \perp$
 - $NewRegion(R) = NumRegions$
 - $NewRegion(NumRegions) = NumRegions$
- ii. Delete Y from region R
- iii. Add Y at $RTAIL(NewRegion(R))$
- iv. $REGION(Y) = NewRegion(R)$
- (c) $Y = PDOM.PARENT(Y)$

Figure 6. Regions algorithm.

Rather than explicitly insert region nodes into the control dependence graph, we find it more convenient to compute regions using the following data structures:

REGION(X): region number associated with node X

RHEAD(R): first node in region R

RTAIL(R): last node in region R

IDNEXT(X): node after X in region $REGION(X)$

IDPREV(X): node before X in region $REGION(X)$

Initially, the above structures are established so that all CFG nodes belong to region 0. All nodes are inserted into the list with head $RHEAD(0)$ by a postorder traversal of the postdominator tree, so that $RTAIL(0) = STOP$ and the postdominator of any node X is linked into the list somewhere after X . At termination, the variable $NumRegions$ contains the number of regions created for the graph. The algorithm is shown in Figure 6. In terms of the postdominator tree, step 3b performs the following tests:

$(Y \succcurlyeq X)$ Is Y a proper ancestor of X in the postdominator tree?

$(Y \succeq X)$ Is Y is an ancestor of X in the postdominator tree?

These tests can be performed in constant time once the nodes of the postdominator tree are depth-first numbered [SS78].

Once regions are identified, $IDENTCD(X)$ is easily computed by traversing a linked list from $RHEAD(REGION(X))$ to $RTAIL(REGION(X))$.

The following easily proved facts are helpful in proving the algorithm correct:

1. Suppose the algorithm processes edges of CFG in the (arbitrary) order $(X_1, L_1), (X_2, L_2), \dots, (X_i, L_i)$. While processing edge (X_i, L_i) , the variable T represents the number of regions created after considering edges $(X_1, L_1), (X_2, L_2), \dots, (X_{i-1}, L_{i-1})$.
2. Throughout the algorithm, $NumRegions$ is maintained as the number of regions assigned to CFG nodes.
3. When a new control condition is processed, nodes belonging to the same region may be split into separate regions. $NewRegion(R)$ contains the most recent such change for region R .
4. For $R \neq 0$ we have

$$IDNEXT^{i+1}(RHEAD(R)) \succcurlyeq IDNEXT^i(RHEAD(R))$$

where $IDNEXT^i$ represents i applications of the $IDNEXT$ mapping. That is, if the nodes in region R are listed from $RHEAD(R)$ to $RTAIL(R)$ by following $IDNEXT()$ pointers, then each node listed is postdominated by the next node listed. For region 0, the postdominator of node $IDNEXT^i(RHEAD(R))$ is not necessarily adjacent, but is $IDNEXT^k(RHEAD(R))$ for some $k > i$.

Theorem 2

1. The algorithm is correct: two CFG nodes X and Y have the same set of control conditions if and only if $REGION(X) = REGION(Y)$.
2. The algorithm in Figure 6 computes $REGION$ $O(N + E)$ space and $O(N \times E)$ time.

Proof:

1. We prove correctness by induction, with the assertion that at step 1, nodes are correctly partitioned into regions with respect to the (X, L) pairs already processed. The base case is trivial: with no control conditions processed, all nodes are in the same region.

Now consider the inductive step. We may assume that with respect to the $i - 1$ control conditions $(X_1, L_1), (X_2, L_2), \dots, (X_{i-1}, L_{i-1})$ already considered by step 1, nodes are correctly partitioned into regions. We must prove that with respect to the control conditions $(X_1, L_1), (X_2, L_2), \dots, (X_i, L_i)$,

$$REGION(Y) = REGION(Z) \iff CONDS(Y) = CONDS(Z)$$

The only nodes whose control dependence predecessors change as a result of considering (X_i, L_i) are those nodes in the interval $[CD.START(X, L), CD.END(X, L))$, because no other nodes are control dependent on (X_i, L_i) . Because the algorithm can affect only those nodes, our proof can be confined to that interval.

\Rightarrow Suppose to the contrary that two nodes have different control conditions. If the difference is found in $(X_1, L_1), (X_2, L_2), \dots, (X_{i-1}, L_{i-1})$, then we already have $REGION(Y) \neq REGION(Z)$. Otherwise, the nodes begin with the same region numbers and the difference must be due to (X_i, L_i) . Assume without loss of generality that node Y is control dependent on (X_i, L_i) while node Z is not. Because node Z is not in the interval $[CD.START(X, L), CD.END(X, L))$, the algorithm will not change $REGION(Z)$. However, node Y is in the interval, but at least one node (Z) in $REGION(Y)$ is not in the interval. We have two possibilities:

$Z \gg Y$: Because Z is in $REGION(Z)$, we have $RTAIL(REGION(Z)) \geq Z$. For Z to be outside the interval we must have

$$CD.END(X_i, L_i) \not\geq RTAIL(REGION(Z))$$

Thus the test at step 3b succeeds and the region number associated with Y is changed.

$Y \gg Z$: Because Z is in $REGION(Z)$, we have $Z \geq RHEAD(REGION(Z))$. For Z to be outside the interval, we must have

$$\begin{aligned} Z &\not\geq CD.START(X_i, L_i) \\ &\text{and} \\ Z &\neq CD.START(X_i, L_i) \end{aligned}$$

which implies

$$\begin{aligned} RHEAD(REGION(Z)) &\not\geq CD.START(X_i, L_i) \\ &\text{and} \\ RHEAD(REGION(Z)) &\neq CD.START(X_i, L_i) \end{aligned}$$

Thus, the test at step 3b succeeds and the region number associated with Y is changed.

In summary, the test at step 3b determines that the region associated with Y is not contained in the interval

$$[CD.START(X_i, L_i), CD.END(X_i, L_i))$$

The region number associated with Y is subsequently changed to a number larger than T . Since Z is outside the interval, its region number is unchanged. After processing (X_i, L_i) , we obtain

$$REGION(Z) \leq T < REGION(Y)$$

\Leftarrow We may assume that with respect to control conditions $(X_1, L_1), (X_2, L_2), \dots, (X_{i-1}, L_{i-1})$, nodes with the same control conditions are already assigned the same

region number. While processing (X_i, L_i) , each node Y in the interval

$$[CD.START(X_i, L_i), CD.END(X_i, L_i))$$

becomes control dependent on (X_i, L_i) . Consider any two nodes Y and Z from that interval such that $REGION(Y) = REGION(Z)$ before processing (X_i, L_i) . We must prove that after considering (X_i, L_i) , we still have $REGION(Y) = REGION(Z)$, even though the region numbers themselves have changed.

There are two possibilities:

1. The set of nodes with region $REGION(Y)$ is contained in $CD(X_i, L_i)$: In this case, the test at step 3b never succeeds, so the region numbers of Y and Z are unchanged.
2. Not all nodes in region $REGION(Y)$ are contained in $CD(X_i, L_i)$. Assume without loss of generality that $Z \geq Y$. When Y is considered at step 3b, the test succeeds and a new region number is assigned to Y . The mapping from the old region associated with Y to the new region is retained in the structure *NewRegions*. Because a region is mapped at most once while considering the control condition (X_i, L_i) , the same mapping is in effect when node Z is considered. Again the test at step 3b succeeds, but we must have $NewRegions(REGION(Z)) > T$, so node Z is assigned the region retained in *NewRegions*. Thus, Y and Z are both changed to the same region number.

2. The $O(N + E)$ space bound follows from the fact that identical region numbers imply identical control conditions (proved above). The number of distinct control conditions is bounded by E . The data structures *RHEAD*, *RTAIL*, and *NewRegion* are of size proportional to the number of regions, while all other data structures are of size proportional to the number of nodes.

The time bound follows from the two loops in the program. The outer loop iterates over the number of edges in *CFG* while the inner loop traverses *PDOM*, an $O(N)$ structure. All other operations take $O(1)$ time, including the \gg tests [SS78]. \square

For the control dependence graph in Figure 2, no two nodes have identical control conditions so each node will be in a region by itself.

5 The Factored Control Dependence Graph

In this section we turn to the question of computing *COND*s efficiently. Unfortunately, this set appears to be more difficult to compute than the other sets we consider here. Previous work [FOW87] created a *factored* control dependence graph with region nodes. Using this graph, the control dependence predecessors (successors) of a node can be enumerated in time proportional to the number of predecessors (successors). Unfortunately, this graph can

use space $O(N \times E)$, and requires for construction worst-case time $O(N^2 \times E)$. While we have not been able to overcome the quadratic space bound, in this section we consider two other factoring methods which may achieve less space in practice. Both methods we present have a worst case time bound of only $O(N \times E)$.

5.1 Factoring by Descendants

In [FOW87], factoring of sets of control conditions is performed only between the sets of a child and its parent in $PDOM$. The first method we consider performs such factoring based on containment between the sets of *descendants* of a given node in $PDOM$. Because it can perform more factoring, this new method may achieve less space in practice. In addition, this new factored graph can be constructed in time $O(N \times E)$, which improves the worst case time of [FOW87] by a factor of N ; this improvement comes from using a fast test for set containment [Yel90] instead of hashing. In Figure 7, we give an example of a loop-free CFG which requires quadratic space with the factoring algorithm of [FOW87] and linear space with the first factoring algorithm presented here. However, like [FOW87], our factored graph can use at worst space $O(N \times E)$.

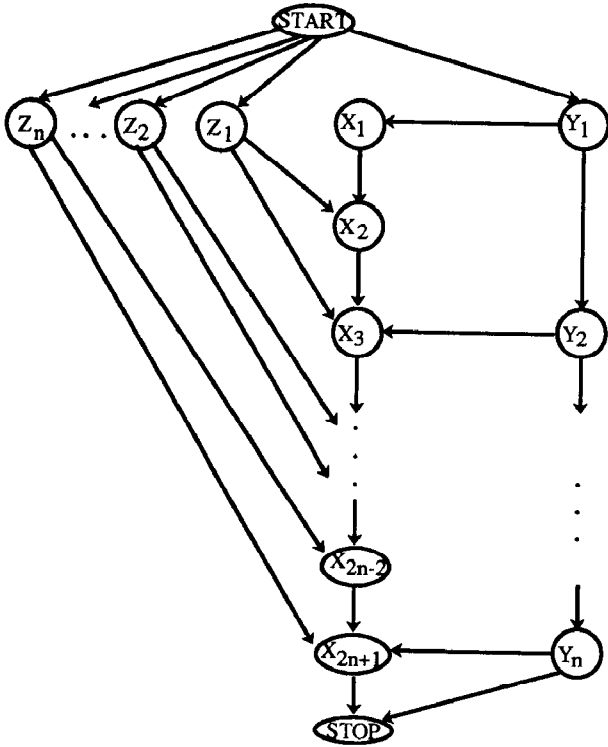


Figure 7. Quadratic space for factoring with children, linear space for factoring with descendants

With the structure determining region nodes already computed, we find it convenient to compute predecessors of regions using the following three data structures:

1. The array $RPREDS(1..R)$, where $RPREDS(A)$ is the set of *control conditions* for region A . A control

condition is a pair (X, L) where X is a node of CFG and L is the label of an edge from X .

2. The array $CPREDS(1..R)$, where $CPREDS(A)$ is the set of region node predecessors for region A .
3. The array $CPREDS(1..R)$, where $CPREDS(A)$ is the set of control condition predecessors for region A .

Given the two data structures $RPREDS$ and $CPREDS$, it is easy to construct a control dependence graph with region nodes factored using $RPREDS$.

Figure 8 contains the algorithm that determines $RCONDS$, $RPREDS$, and $CPREDS$. We assume that the array $REGION$ has already been computed as in Section 4. Step 3 contains the bottom-up traversal of $PDOM$. The data structures $COMMON$ and INV_RCONDS are maintained so that a set containment test can be performed in constant time in steps 3d and 3f [Yel90]. $COMMON[i, j]$ keeps track of the number of common elements between the sets $RCONDS(REGION(i))$ and $RCONDS(REGION(j))$. For each control condition (X, L) , $INV_RCONDS[(X, L)]$ contains the set of nodes i such that $(X, L) \in RCONDS(REGION(i))$. Note that $COMMON$ is a symmetric matrix, which could be represented by one of its triangular halves (see [Yel90] for more details).

Theorem 3

1. The algorithm in Figure 8 computes $RCONDS$, $RPREDS$, and $CPREDS$ in $O(N \times E)$ space and time.
2. $CONDS(X) = RCONDS(REGION(X))$.
3. For any node X ,

$$RCONDS(REGION(X)) = CPREDS(REGION(X)) \cup \bigcup_{R \in RPREDS(REGION(X))} RCONDS(R)$$

Proof:

1. The $O(N \times E)$ space bound follows from the fact that the largest data structures computed by the algorithm are $RCONDS$, $RPREDS$, $CPREDS$, $COMMON$ and INV_RCONDS , all of which can occupy $O(N \times E)$ space in the worst case.

To establish the $O(N \times E)$ time bound, we assume that the sets of control conditions are represented as sorted lists, so that all following set operations can be performed in linear time: *union*, *intersection*, *difference*. Since a set may have $O(E)$ elements, each operation will take $O(E)$ time. Each of the set operations within step 3 is performed $O(N)$ times, thus giving a total contribution of $O(N \times E)$. This is true even for the set union over node G 's children in steps 3b, and 3e, since $PDOM$ is a tree.

We have to be a little more careful in step 1, new elements are inserted into a set. If insertion is naively implemented as set union, the total execution time for step 1 could be $O(E^2)$ in the worst case. However, we assume that the **for** loop steps through

control conditions in the same order as that used for maintaining the sorted lists. Therefore, in this case, set insertion can be performed by a simple *append* operation which takes $O(1)$ time, so that the total execution time for step 1 is $O(E)$.

The search for $D(C)$ for each child C of G requires a search of G 's subtree and a constant time check at each node in the subtree. Thus over all nodes in the tree, step 3d contributes time $O(N^2)$.

2. We first observe that step 1 correctly computes the following sets. $START(i)$ and $END(i)$ for each node i :

$START(i)$ = set of control conditions, (X, L) , such that

$$\begin{aligned} CD.START(X, L) &= i \\ &\text{and} \\ CD.START(X, L) &\neq CD.END(X, L) \end{aligned}$$

$END(i)$ = set of control conditions, (X, L) , such that

$$\begin{aligned} CD.END(X, L) &= i \\ &\text{and} \\ CD.START(X, L) &\neq CD.END(X, L) \end{aligned}$$

We then establish an invariant at the beginning of each iteration of the **for** loop in step 3, stating that $RCONDS$ has been correctly computed for all of G 's descendant nodes in $PDOM$. This invariant is trivially true initially, since the **for** loop must begin with a leaf node in $PDOM$. For each child, C , of G in $PDOM$, step 3b combines the $RCONDS(REGION(C))$ sets to correctly compute P , the set of G 's control conditions.

Now, the only question that remains is whether P is the same as any of the previously computed sets in $RCONDS$. This question is answered by simply consulting $REGION$ (step 3).

- 3.

(\subseteq): Consider any

$$(X, L) \in RCONDS(REGION(X))$$

If $(X, L) \in CPREDS(REGION(X))$, we are done. So suppose not. The only

$$(X, L) \in RCONDS(REGION(X))$$

excluded from $RPREDS(REGION(X))$ are those in $RCONDS(REGION(R))$ for some child $R \in RPREDS(REGION(X))$.

(\supseteq): If

$$(X, L) \in CPREDS(REGION(X))$$

then by construction

$$(X, L) \in RCONDS(REGION(X))$$

So suppose $(X, L) \in RCONDS(R)$, where $R \in RPREDS(REGION(X))$. Then by construction, $R = REGION(D)$, for some descendant D of a child C of X , and $COMMON[D, D] = COMMON[X, X]$. But then it must be the case that

$$RCONDS(REGION(D)) \subseteq RCONDS(REGION(X))$$

so $(X, L) \in RCONDS(REGION(X))$.

□

5.2 Factoring Using ADD Sets

Recall from the previous section that for a node G ,

$$CONDS(G) = \left(\bigcup_{C \in children(G)} (CONDS(C) - END(G)) \right) \cup START(G)$$

where $CONDS(X) = RCONDS(REGION(X))$. We define, for each $C \in children(G)$

$$ADD(C) = CONDS(C) - END(G)$$

Intuitively, $ADD(C)$ is child C 's contribution to $CONDS(G)$.

The second method for constructing a factored graph is to split the construction into two steps. We create regions for each new ADD set. Since $CONDS(G)$ is the union of $START(G)$ and the $ADD(C)$ sets of its children, $RPREDS(REGION(G))$ is made to consist of the regions of $ADD(C)$ for each $C \in children(G)$; $CPREDS(REGION(G))$ will be the elements of $START(G)$. We then compute and represent $ADD(G)$, factoring it as in Section 5.1 in terms of G 's descendants. $RPREDS$ thus has two layers of factoring, in terms of the ADD and $START$ sets, and in terms of descendants. Although we increase the number of regions by representing the ADD sets explicitly, we hope that overall the increased factoring will decrease the amount of space used. Figure 9 shows the factored control dependence graph obtained for the control flow graph in Figure 1. Names of the form R_x and R' are used for region nodes only.

In contrast, in [FOW87] the $ADD(C)$ sets were not represented, and factoring only took place when $CONDS(C) \cap CONDS(G)$ equalled one of the sets. For example, the algorithm in [FOW87] would perform no factoring on the control dependence predecessor sets of nodes E and F in Figure 2, but these sets will be factored by our algorithm. It is this factoring in terms of ADD sets which differentiates us from [FOW87]. In Figure 10, we give an example of a loop-free control flow graph which requires quadratic space with both the factoring algorithms of [FOW87] and Section 5.1, and linear space with the factoring algorithm presented in this section.

The algorithm sketched in Figure 11 is similar to that in Figure 8, and is a bottom-up traversal of $PDOM$. We point out here several differences between the first

```

1. /* Compute  $START(1 \dots N)$  and  $END(1 \dots N)$  */
    $START(*) := \phi$  ;  $END(*) := \phi$ 
   for each control condition  $(X, L)$  such that  $CD.START(X, L) \neq CD.END(X, L)$  do
     Add  $(X, L)$  to the sets  $START(CD.START(X, L))$  and  $END(CD.END(X, L))$ 
   end for
2.  $COMMON[1 \dots N, 1 \dots N] := 0$  ;  $INV\_RCONDS[*] = \phi$  ;
3. for each node  $G$  such that each child  $C$  of  $G$  in  $PDOM$  has been visited and  $(G = RHEAD(REGION(G)))$  do
  (a) Mark  $G$  as visited
     /* $G$  is the start of a new region, create entries for  $RCONDS$ ,  $RPREDS$ , and  $CPREDS$ .*/
  (b)  $P := START(G) \cup \bigcup_{C \in children(G)} RCONDS(REGION(C)) - END(G)$ ;
     /* Now  $P =$  set of  $G$ 's control conditions */
  (c) /* Update  $COMMON$  and  $INV\_RCONDS$  */
     for each  $(X, L) \in P$  do
       i. Insert  $G$  in the set  $INV\_RCONDS[(X, L)]$  ;
       ii.  $COMMON[G, G] := COMMON[G, G] + 1$  ;
       iii. for each  $i \in INV\_RCONDS[(X, L)]$  such that  $i \neq G$  do
          $COMMON[G, i] := COMMON[G, i] + 1$  ;  $COMMON[i, G] := COMMON[i, G] + 1$  ;
       end for
     end for
  (d) /* Find a descendant of each child with maximal contained set*/
     for each child  $C$  of  $G$  do
       Let  $D(C)$  be a descendant  $D$  of  $C$  such that
        $COMMON[D, G] = COMMON[D, D]$  and  $D(C)$  has the largest number of common elements with  $G$  over all
       descendants  $D$  of  $C$  with that property.
  (e) /* Fill in  $RCONDS$ ,  $RPREDS$  and  $CPREDS$ */
       i.  $RCONDS(REGION(G)) := P$ 
       ii.  $RPREDS(REGION(G)) := \bigcup_{children\ C\ of\ G} \{D(C)\}$ 
       iii.  $CPREDS(REGION(G)) := (P - \bigcup_{children\ C\ of\ G} \{RCONDS(REGION(D(C)))\})$ .

     end for
  (f) /* Find a descendant of each child with maximal containing set*/
     for each child  $C$  of  $G$  do
       Let  $D(C)$  be a descendant  $D$  of  $C$ , if it exists, such that
        $COMMON[D, G] = COMMON[G, G]$  and  $D(C)$  has the largest number of common elements with  $G$  over all
       descendants  $D$  of  $C$  with that property.
  (g) if  $D(C)$  exists, /* Fill in  $RPREDS$  and  $CPREDS$  */
       i.  $RPREDS(REGION(D(C))) := RPREDS(REGION(D(C))) \cup \{REGION(G)\}$ 
       ii.  $CPREDS(REGION(D(C))) := CPREDS(REGION(D(C))) - CPREDS(REGION(G))$ 

     end for
end for

```

Figure 8. Algorithm: Factoring with respect to containment of descendants

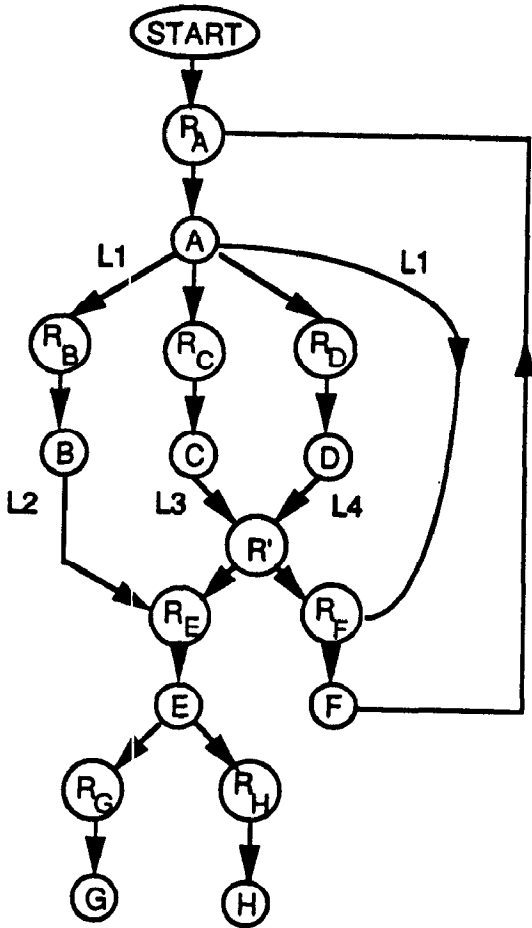


Figure 9. Factored graph

and second factoring algorithms. First, we create region numbers for new *ADD* sets as they are constructed. To keep track of when new sets are created, we extend the structure *COMMON* so that indices $1 \leq i \leq N$ correspond to *CONDS* sets as before, and indices $N + 1 \leq i \leq 2 * N$ correspond to *ADD* sets.

As before, new region numbers are created only if no other region node represents its set of control conditions. For a node *G*, whether *CONDS*(*G*) is new is determined by the predicate *NEWCONDS*(*G*) and whether *ADD*(*G*) represents a new set is determined by the predicate *NEWADD*(*G*). We define:

1. *NEWCONDS*(*G*) is true iff *CONDS*(*G*) \neq *CONDS*(*D*) and *CONDS*(*G*) \neq *ADD*(*D*) for any descendant *D* of *G* in *PDOM*.
2. *NEWADD*(*G*) is true iff *ADD*(*G*) \neq *CONDS*(*G*), *ADD*(*G*) \neq *CONDS*(*D*) and *ADD*(*G*) \neq *ADD*(*D*) for any descendant *D* of *G* in *PDOM*.

Both of these predicates are determined by using the *COMMON* structure.

Lemma 2 *NEWCONDS*(*G*) and *NEWADD*(*G*) can be computed in $O(N)$ time using the structure *COMMON*.

Proof:

1. $CONDS(X) = CONDS(Y) \iff$
 $COMMON[X, X] = COMMON[X, Y]$

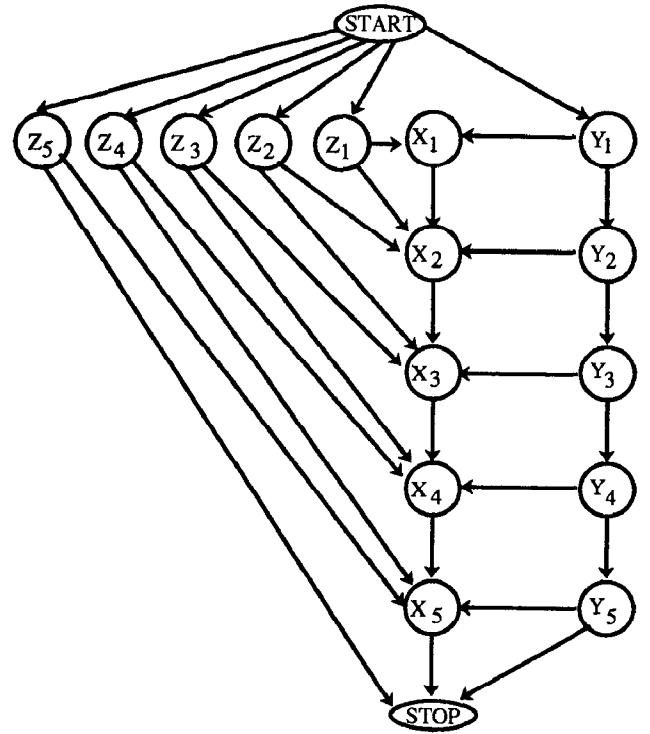


Figure 10. Quadratic space with descendants, linear space with *ADD* sets

$$= COMMON[Y, Y]$$

2. $CONDS(X) = ADD(Y) \iff$

$$COMMON[X, X] = COMMON[X, Y + N]$$

$$= COMMON[Y + N, Y + N]$$

3. $ADD(X) = ADD(Y) \iff$

$$COMMON[X + N, X + N]$$

$$= COMMON[X + N, Y + N]$$

$$= COMMON[Y + N, Y + N]$$

For any node *G*, we can test all descendants by a walk through *G*'s subtree and a constant time test at each node. \square

Theorem 4

1. The algorithm in Figure 11 computes *RPREDS* and *CPREDS* in $O(N \times E)$ space and time.
2. For any node *X* in *CFG*,

$$CONDS(X) = CPREDS(REGION(X)) \cup$$

$$\bigcup_{R \in RPREDS(REGION(X))} RCONDS(R)$$

Proof:

1. The $O(N \times E)$ space bound follows from the fact that *COMMON*, *RPREDS* and *CPREDS* are of at most that size. The execution time of our algorithm is dominated by the enumeration, for every node *G*, of the subtree in *PDOM* rooted at *G*, and a constant time check at each node.

```

1.  $COMMON[1 \dots 2 * N, 1 \dots 2 * N] := 0$  ;
2. for each node  $G$  such that each  $C \in children(G)$  has been visited do
  (a) Mark  $G$  as visited.
  (b)  $CONDS(G) \leftarrow (\bigcup_{C \in children(G)} ADD(C)) \cup START(G)$ .
  (c) if  $NEWCONDS(G)$  then
    i. Create a new region number  $REGION(G)$ , and update  $COMMON$  for  $G$ .
    ii.  $RPREDS(REGION(G)) := \bigcup_{C \in children(G)} \{AREGION(C)\}$ .
    iii.  $CPREDS(REGION(G)) := START(G)$ .
  (d) if  $G \neq STOP$  then
    i.  $ADD(G) = CONDS(G) - END(P)$ .
    ii. if  $NEWADD(G)$  then
      A. Create a new region number  $AREGION(G)$ , and update  $COMMON$  for  $G + N$ .
      B. /* Find a descendant of each child with maximal contained set for  $ADD(G)$ */
         for each  $C \in children(G)$  do
           Let  $RD(C)$  be a descendant  $D$  of  $C$  such that  $COMMON[D, G + N] = COMMON[D, D]$  and  $RD(C)$  has
           the largest number of common elements with  $ADD(G)$  over all descendants  $D$  of  $C$  with that property.
           Let  $AD(C)$  be a descendant  $D$  of  $C$  such that  $COMMON[D + N, G + N] = COMMON[D + N, D + N]$ 
           and  $AD(C)$  has the largest number of common elements with  $ADD(G)$  over all descendants  $D$  of  $C$  with that
           property.
           if  $COMMON[RD(C), G + N] > COMMON[AD(C) + N, G + N]$ 
             then  $R(C) := REGION(RD(C))$  ;  $SET(C) := CONDS(RD(C))$ 
           else  $R(C) := AREGION(AD(C))$ ;  $SET(C) := ADD(AD(C))$ 
           end for
      C. /* Fill in  $RPREDS$  and  $CPREDS$ */
          $RPREDS(REGION(G)) := \bigcup_{C \in children(G)} \{R(C)\}$ 
          $CPREDS(REGION(G)) := (ADD(G) - \bigcup_{C \in children(G)} \{SET(C)\})$ .
      D. /* Find a descendant of each child with maximal containing set for  $ADD(G)$ */
         for each  $C \in children(G)$  do
           Let  $RD(C)$  be a descendant  $D$  of  $C$ , if it exists, such that  $COMMON[D, G + N] = COMMON[G + N, G + N]$ 
           and  $RD(C)$  has the largest number of common elements with  $ADD(G)$  over all descendants  $D$  of  $C$  with that
           property.
           Let  $AD(C)$  be a descendant  $D$  of  $C$ , if it exists, such that  $COMMON[D + N, G + N] = COMMON[G + N, G + N]$ 
           and  $AD(C)$  has the largest number of common elements with  $ADD(G)$  over all descendants  $D$  of
            $C$  with that property.
           if both  $RD(C)$  and  $AD(C)$  exist then if  $COMMON[RD(C), G + N] > COMMON[AD(C) + N, G + N]$ 
             then  $D(C) := RD(C)$  else  $D(C) := AD(C)$ 
           else if only one exists, then  $D(C)$  is given it as value; else  $D(C) := \perp$ 
      E. if  $D(C) \neq \perp$  then /* Fill in  $RPREDS$  and  $CPREDS$  */
          $RPREDS(D(C)) := RPREDS(D(C)) \cup \{AREGION(G)\}$ 
          $CPREDS(D(C)) := CPREDS(D(C)) - CPREDS(AREGION(G))$ 
         end for
    end for
  end for

```

Figure 11. Algorithm: Factoring with respect to ADD sets

2. Follows from Theorem 3.

□

Lemma 3 *The time to enumerate the control dependence predecessors of a node X is at most twice the number of control dependence predecessors of X .*

Proof: (sketch) Define for some i

$$Ranc(R) = \{Z \mid Z \in RPREDSt(REGION(X))\}$$

where $RPREDSt$ denotes i applications of $RPREDSt$. Define

$$Ianc(R) = \{(X, L) \in CPREDSt(Z) \mid Z \in Ranc(R)\}$$

We show by induction, starting from leaf nodes in $PDOM$, that

$$|Ranc(R_G)| \leq 2 \times |Ianc(G)|$$

for any region number R_G associated with the node G . In the base case, $Ianc(R_G) = START(G)$, and the result follows since $START(G)$ has at least one element. Assume that the induction hypothesis holds for each $C \in children(G)$. By construction,

$$Ranc(R_{CONDS(G)}) = \left(\bigcup_{C \in children(G)} Ranc(R_{ADD(C)}) \right)$$

So,

$$|Ranc(R_{CONDS(G)})| = \left(\sum_C |Ranc(R_{ADD(C)})| \right)$$

But by the induction hypothesis,

$$\left(\sum_C |Ranc(R_{ADD(C)})| \right) \leq 2 \times \left(\sum_C |Ianc(C)| \right)$$

Since all the sets $Ianc(C)$ are disjoint, we have

$$2 \times \left(\sum_C |Ianc(C)| \right) \leq 2 \times |Ianc(G)|$$

□

6 Empirical Results

In this section we summarize the results of experiments performed to measure the success of our methods for determining $CD(X, L)$, $IDENTCD(Y)$, and $CONDS(Y)$. Programs were chosen for our experiments from large, popular numerical analysis benchmarks written in (unstructured) Fortran. Over 400 procedures were analyzed, with a total of 28413 statements. Each procedure contained from 7 to 753 statements, with the average (median) procedure containing 31 statements. The average number of statements in a procedure was 69. Although our experiments were performed using the PTRAN system [ABC*87], the numbers we report are *not* sizes of data structures within PTRAN, because these are language-dependent. Instead, we develop a model for the number of words required to store such structures, where a word must be sufficiently large to contain an integer or a pointer. Also, we did not add the usual interval (loop) entry or exit nodes to the

control flow graph, because not all systems that use control dependence require interval structures. For at least the case of nested **repeat...until** constructs, augmenting a control flow graph with the proper interval nodes [CFS89] avoids quadratic behavior.

First, consider how the full control dependence graph might be organized, to the extent required to compute $CD(X, L)$. We assume that the edges of the control flow graph CFG are numbered from $1 \dots |E_{CFG}|$. In a full control dependence graph representation, we would represent $CD(X, L)$ as a linked list of nodes, each control dependent on X due to edge L . We also require an array of $|E_{CFG}|$ pointers to specify the beginning of each list. Using the algorithm of Figure 5 to compute the lists, we see that an entry is created for each node contained in the interval $[CD.START(X, L), CD.END(X, L)]$. The total number of entries is therefore

$$|E_{CDG}| = \sum_{(X, L) \in E_{CFG}} \binom{\text{depth}(CD.START(X, L)) - \text{depth}(CD.END(X, L))}{1}$$

where $depth(Z)$ is the distance of node Z from the root of the postdominator tree. Using a model where data and pointers each occupy one word of storage, the number of words required for this representation is:

$$|E_{CFG}| + 2 \times |E_{CDG}|$$

Using the methods described in Section 3, $CD(X, L)$ can be computed without any extra storage. If only one CD set is required at a time, then our methods require no storage beyond the control flow graph and postdominator tree. For the 410 programs we analyzed, the total space saved was 110502 words, but CD information is typically computed for one procedure at a time. The procedure with the largest full control dependence graph was *MOSFET* (from the *SPICE* package), but it was not the program with the most (executable) statements. In fact, the program *BJT* was twelfth largest in size, yet had the third largest control dependence graph. Consider as a measure of linearity the ratio between the control dependence graph size and the number of executable statements in a program. Table 1 shows number of storage words required to store the full control dependence graphs for the 20 programs with the largest such ratio. The median procedure required a graph whose size (in words) was 3.4 times its number of statements.

Now consider how to compute $IDENTCD(X)$.

1. Regions nodes can be inserted [FOW87] into the full control dependence graph, resulting in a graph with potentially fewer edges (but probably more nodes). $IDENTCD(X)$ can then be computed by listing all control dependence graph successors of node $REGION(X)$.

The $REGION$ structure requires $|N_{CFG}|$ words. If a control dependence graph with $|E_{CDG}|$ edges is organized by $NumRegions$ regions, then the resulting graph has $|N_{CFG}| + NumRegions$ nodes. There are $|N_{CFG}|$ edges that connect a CFG node with its region node. For each region R , there are

Program Name	Number of Statements	Words to Store Full CD Graph	Ratio
MOSFET	623	3742	6.0
BJT	433	2285	5.3
MATINV	116	602	5.2
QCD2	191	991	5.2
MULT	38	193	5.1
MTINV	63	316	5.0
MAIN2	544	2723	5.0
JFET	280	1398	5.0
DFLUX	162	806	5.0
FITPAR	104	506	5.0
DFLUXC	75	364	4.9
SOLXDD	96	465	4.8
PSMOO	61	294	4.8
EFLUX	61	294	4.8
COLLC	54	259	4.8
INTPL	37	176	4.8
REORDR	89	423	4.8
FOURAN	72	342	4.8
COMCOF	72	341	4.7
ADDX	67	315	4.7

Table 1. Programs with the most nonlinear control dependence graphs

$|CONDS(RHEAD(R))|$ edges that connect the control conditions with the region node. The space required is then

$$|N_{CFG}| + NumRegions + 2 \times \left(|N_{CFG}| + \sum_{r=1}^{NumRegions} |CONDS(RHEAD(r))| \right)$$

- Using the algorithm of Figure 6, the *REGIONS*, *RHEAD*, and *RTAIL* structures can be created, taking space $|N_{CFG}|$, $NumRegions$, and $NumRegions$, respectively. The *IDNEXT* and *IDPREV* pointers each take $|N_{CFG}|$ storage. The total space required for this representation is

$$2 \times NumRegions + 3 \times |N_{CFG}|$$

Again we considered the 20 procedures with the largest ratio of control dependence graph size to executable statements; this time, the graph measurements included space for region nodes and their associated edges. In Table 2, we show the procedures sorted by this ratio (the ratio itself is not shown, but is easily computed from the second and third columns). We also show the number of words for storing the same information (with respect to *CD* and *IDENTCD*) using our methods, and the last column shows the percentage improvement. The median program experienced an 11% improvement, while the best improvement was experienced by the second worst program *QCD*.

Although the benchmarks contained over 300 procedures, we now consider only the largest 100 procedures to examine how our methods save space in representing

Program Name	#Stmts	Graph with Regions	Our Region Structs.	Percent Better
INDXX	29	153	119	28.6
QCD2	191	999	681	46.7
FNDNAM	40	209	166	25.9
CARD	234	1222	954	28.1
OUTDEF	118	591	476	24.2
MULT	38	189	144	31.3
ALTER	78	387	316	22.5
RUNCON	633	3094	2537	22.0
SHAPE	24	117	102	14.7
READIN	753	3660	2917	25.5
TERR	49	236	181	30.4
UPDATE	29	139	111	25.2
MATINV	116	554	436	27.1
PLOT	12	57	50	14.0
TOPCHK	187	878	723	21.4
ITER8	94	440	366	20.2
QUAD	29	135	119	13.4
FIND	78	363	308	17.9
ACCE	18	83	72	15.3
ADDELT	79	364	307	18.6

Table 2. Improvement for graphs with regions identified.

CONDS(Y). These procedures contained from 70 to 753 statements, with the average (median) procedure containing 130 statements. Recall that we could not obtain a linear bound for computing *CONDS*. Thus, it is especially interesting that the plot shown in Figure 12 shows linear behavior for these programs. The points labelled with an "N" show the size of the full control dependence graph with regions, from which *CONDS* is easily computed. Although the apparently linear behavior leaves little room for improvement, we also show points labelled "C", "D", and "A", which show the size of the graphs when improvement occurs due to factoring by child [FOW87], by descendant (Section 5), or by *ADD* set (Section 5.2), respectively.

In summary, we improve previous work by guaranteeing linear behavior if only the *CD* and *IDENTCD* aspects of the control dependence graph are required. If *CONDS* is needed, then we have presented methods that address several, but not all, causes of quadratic behavior. However, the space required to represent even the full control dependence graph appears linear, even for large, unstructured procedures. Thus, in practice, our methods cannot substantially improve the space required to compute or store *CONDS*.

7 Open Questions and Future Work

Unfortunately, being able to answer the question

On what nodes is *Y* control dependent?

appears to be more difficult than the other questions we posed in this paper. In particular, we have *not* been able to show that there is a linear space data structure to

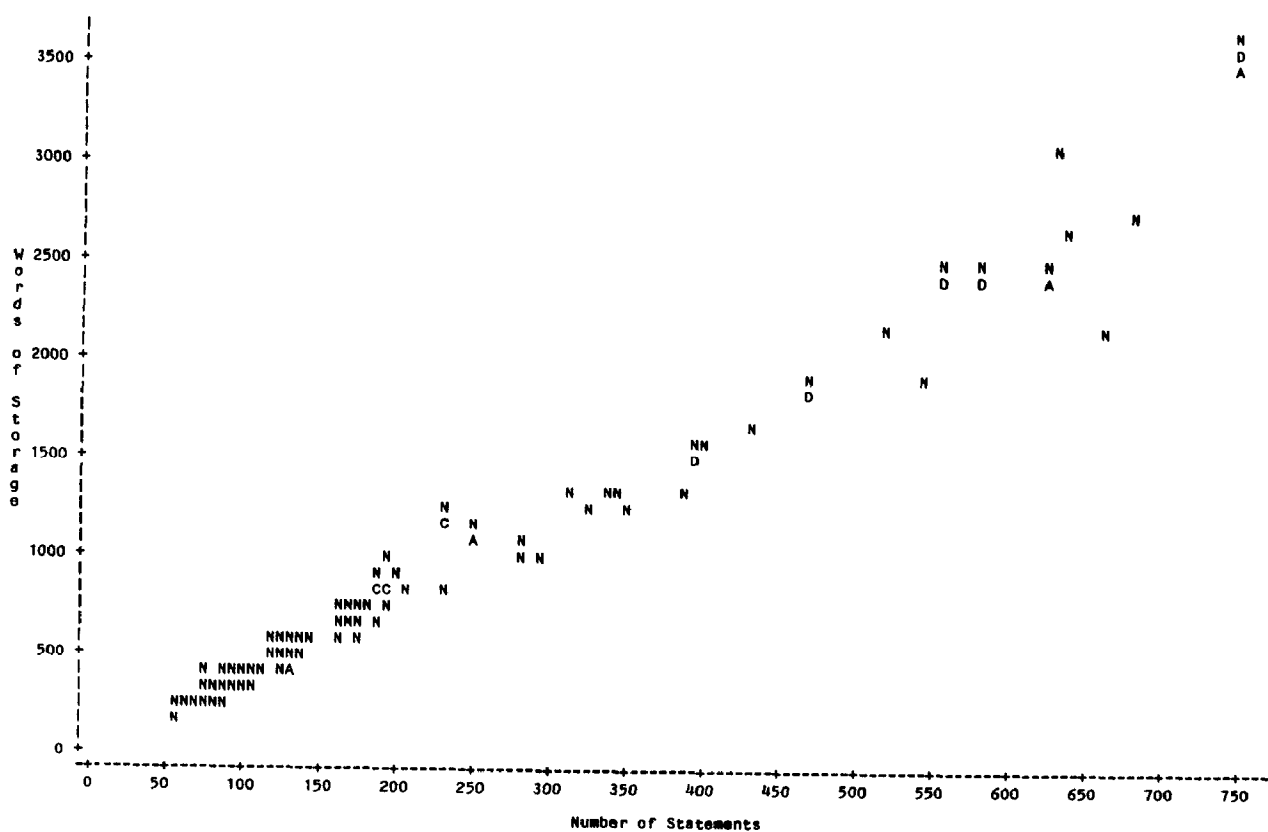


Figure 12. Control dependence storage vs. Program size

answer all such questions that in addition only uses time proportional to the size of each answer. The data structure of Section 3 uses only linear space and allows us to answer such questions, but the time needed may be $O(N)$, even if a node has a single predecessor. The data structure of Section 5 allows us to answer such questions in time proportional to the answer, but may use quadratic space.

This question is particularly motivated by the relationship of control dependence to SSA form. SSA form is a program representation which ensures that there is a single assignment to each variable (statically), and distinguishes the values of variables transmitted on distinct incoming control flow edges. The advantages of these properties have been exploited in [WZ85, CLZ86, CF87, AWZ88, RWZ88]. In [CFR*89], it was shown how to compute SSA efficiently from a data structure called *dominance frontiers*. In that work, it was also shown that the dominance frontier relation is given by the control dependence *predecessors* on the *reverse* control flow graph. Therefore, an improved space bound for enumerating control dependences predecessors would beneficially affect the space bound for dominance frontiers, and therefore, for SSA.

Examination of examples for which the control dependence relation requires quadratic space suggests that factoring of control dependence sets is needed both in order of where elements of the sets *start* and where they *end*. First consider the example in Figure 3 where a chain of elements X_i in *PDOM* have as their set of control dependence predecessors the set $\{Y_1 \dots Y_i\}$. At node X_{i+1} where we want to incorporate the control dependence Y_{i+1} , we wish to factor the representation of the already constructed set $\{Y_1 \dots Y_i\}$ into the representation for X_{i+1} . That is, the sets we construct as we go up the tree can be represented

by a new region node that points to both the element that starts at the node and the already constructed set. This kind of factoring is incorporated into our work and [FOW87].

Now consider a similar example, where X_i is control dependent on $\{Y_1 \dots Y_{N-i+1}\}$, and X_{i+1} immediately post-dominates X_i , $1 \leq i < N$. In this case, it makes sense to represent the set $\{Y_1 \dots Y_N\}$ from the beginning so that Y_N, Y_{N-1}, \dots can be successively eliminated from our successive representations of control dependence predecessor sets by simply changing a pointer. That is, the initial set for X_1 is represented by creating a region node for each Y_i , giving this region node Y_i as predecessor, and in addition, if $i < N$, give it the additional predecessor of the region node of Y_{i-1} . Thus an initial *START* set should be represented by factoring it in order of its elements *END* positions in *PDOM*. Without such factoring, all known methods use quadratic space. If *START* sets were factored as well, then this example does not require quadratic space.

While such representations can be dually constructed as one processes a single branch in *PDOM*, a difficulty arises when we get to a *join* of two or more branches. Here, merging the representations we have constructed so far for each branch may require insertions of one representation into another, thereby destroying the representation for each branch. If we do not merge the representations for each branch, we need to duplicate part of the representations, and so leave ourselves open to the use of quadratic space.

We therefore conjecture that to enumerate control dependence predecessors in time proportional to the number of predecessors requires a data structure of quadratic space.

Acknowledgements

We would like to thank Fran Allen, Michael Burke, and other members of the PTRAN group, past and present. We would like to acknowledge Larry Carter for several helpful technical discussions. We would like to thank Paul Havlak of Rice University for comments on an earlier version of this paper.

References

- [ABC*87] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the ptran analysis system for multiprocessing. *Proceedings of the 1987 International Conference on Supercomputing*, 1987. Also published in *The Journal of Parallel and Distributed Computing*, Oct., 1988, Vol. 5, No. 5, pp. 617-640.
- [ABC*88] Frances Allen, Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, and Vivek Sarkar. A framework for determining useful parallelism. *Proceedings of the ACM 1988 International Conference on Supercomputing*, 207-215, July 1988.
- [AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. *Fifteenth ACM Principles of Programming Languages Symposium*, 1-11, January 1988. San Diego, CA.
- [BB89] William Baxter and J. R. Bauer, III. The program dependence graph in vectorization. *Sixteenth ACM Principles of Programming Languages Symposium*, 1 - 11., January 11-13 1989. Austin, Texas.
- [CF87] Ron Cytron and Jeanne Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. *Proceedings of the 1987 International Conference on Parallel Processing*, 19-27, August 1987.
- [CF89] Robert Cartwright and Mathias Felleisen. The semantics of program dependence. *SIGPLAN '89 Conference on Programming Language Design and Implementation*, 13-27, June 1989.
- [CFR*89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 25-35, January 1989.
- [CFS89] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Experiences using control dependence in ptran. *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.
- [CLZ86] Ron Cytron, Andy Lowry, and Ken Zadeck. Code motion of control structures in high-level languages. *Conf. Rec. of the ACM Symp. on Principles of Compiler Construction*, 1986.
- [FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 319-349, July 1987.
- [GS87a] Rajiv Gupta and Mary Lou Soffa. A reconfigurable lww architecture and its compiler. *Proc. of the 1987 Int'l Conf. on Parallel Processing*, Aug. 1987.
- [GS87b] Rajiv Gupta and Mary Lou Soffa. Region scheduling. *Proc. of the Second International Conference on Supercomputing*, 3:141-148, May 1987.
- [Har85] Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. *Symposium on Theory of Computing*, May 1985.
- [HPR87] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *Conf. Rec. Fifteenth ACM Symposium on Principles of Programming Languages*, 133-145, January 1987.
- [Kuc78] David J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, 1978.
- [OBM90] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. Gated single-assignment form: dataflow interpretation for imperative languages. *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, June 1990.
- [RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. *Fifteenth ACM Principles of Programming Languages Symposium*, 12-27, January 1988. San Diego, CA.
- [Sel89] Rebecca Parsons Selke. A rewriting semantics for program dependence graphs. *Sixteenth ACM Principles of Programming Languages Symposium*, January 11-13 1989. Austin, Texas.
- [SS78] J. T. Schwartz and M. Sharir. *Tarjan's fast interval finding algorithm*. Technical Report, Courant Institute, New York University, 1978. SETL Newsletter Number 204.
- [WZ85] Mark Wegman and Ken Zadeck. Constant propagation with conditional branches. *Conf. Rec. Twelfth ACM Symposium on Principles of Programming Languages*, 291-299, January 1985.
- [Yel90] Daniel Yellin. Representing sets with constant time equality testing. *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, 64-73, January 1990.