Nesting of Reducible and Irreducible Loops

PAUL HAVLAK
Rice University

Recognizing and transforming loops are essential steps in any attempt to improve the running time of a program. Aggressive restructuring techniques have been developed for single-entry (reducible) loops, but restructurers and the dataflow and dependence analysis they rely on often give up in the presence of multientry (irreducible) loops. Thus one irreducible loop can prevent the improvement of all loops in a procedure. This article gives an algorithm to build a loop nesting tree for a procedure with arbitrary control flow. The algorithm uses definitions of reducible and irreducible loops which allow either kind of loop to be nested in the other. The tree construction algorithm, an extension of Tarjan's algorithm for testing reducibility, runs in almost linear time. In the presence of irreducible loops, the loop nesting tree can depend on the depth-first spanning tree used to build it. In particular, the header node representing a reducible loop in one version of the loop nesting tree can be the representative of an irreducible loop in another. We give a normalization method that maximizes the set of reducible loops discovered, independent of the depth-first spanning tree used. The normalization requires the insertion of at most one node and one edge per reducible loop.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Strongly-connected regions, reducible loops

1. INTRODUCTION

Computer programs do not execute for very long without repeating instructions. To make a program run faster one must find and accelerate the blocks of repetitive code known as *loops*. Most general-purpose procedural languages have special constructs for representing loops, but most also allow arbitrary transfers of control using GOTOs. Speeding up a program can require identifying loops in arbitrary control-flow graphs.

Dealing with messy loops is a problem for many compilers, particularly optimizing compilers for superscalar and parallel computers. Their techniques for aggressively analyzing dependences and restructuring loops commonly focus on handling *reducible* programs — those with only single-entry loops. When reducibility is de-

This research was conducted at Rice University and at the University of Maryland and was supported by the NSF and the Center for Research on Parallel Computation through grants ASC-9213821 and CCR-9120008.
Author's address: Department of Computer Science, Rice University, 6100 South Main Street, Houston, Texas 77005-1892.

```
program main
α: if P goto δ
β: /* ... */
γ: if Q goto κ
δ: if R goto γ
θ: goto δ
κ: on I goto β, λ, κ
λ: if S goto α
end main
```

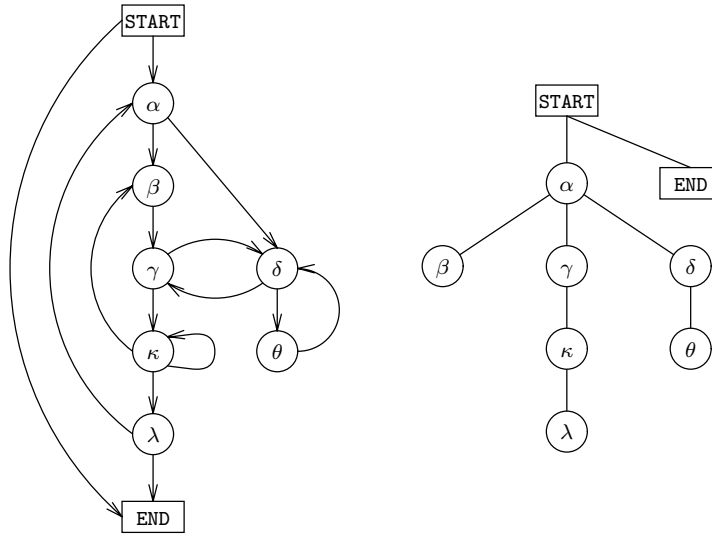

Fig. 1. Example code, G_{CF}, and dominator tree.

sired, but not enforced by language restrictions, it must be checked [Tarjan 1974]. Any irreducible portions must then be ignored (and therefore not optimized) or repaired by the replication of statements. Repair can cause exponential growth in the number of program statements [Aho et al. 1986].

This article presents methods for isolating the irreducible parts of a program and maximizing the portion that can be handled using reducible-loop techniques. Using traditional notions of reducibility and defining notions of nested reducible and irreducible loops, we show how to build a generalized loop nesting tree, with the procedure *analyze_loops* in Figure 5. For purposes of familiarity and efficiency, we have expressed this algorithm as an adaptation of Tarjan's algorithm for testing reducibility [Tarjan 1974].

Our loop nesting tree is not unique when the single entry of a reducible loop can also be an entry of an irreducible loop. Procedure *fix_loops* in Figure 6 adds null statements in such cases to guarantee that a maximal and unique set of reducible loops will be found by *analyze_loops*.

2. BACKGROUND

2.1 Control-Flow Graphs

The control-flow graph $G_{CF} = (N_{CF}, E_{CF})$ is a basic tool in the static analysis of programs. Each node $n \in N_{CF}$ represents a basic block of zero or more executable statements in straight-line sequence, with no control-flow jumps into the block except before the first statement and no jumps out except after the last. Each edge $(s, d) \in E_{CF}$ represents a potential transfer of control after execution of the last statement in the source node s to the first statement in the destination node d.

Figure 1 sketches an example program in pseudocode and gives its control-flow graph G_{CF} (and its dominator tree, described in the next section). The distinguished node START has no incoming edges and has outgoing edges to each node

```
static current := 1
number[*] := UNVISITED      // arbitrary nil value ≤ 0
DFS(START)

procedure DFS(a)
    node[current] := a                  // record nodes visited in order
    number[a] := current++              // record preorder number of this node and advance count
    foreach node b such that ∃ an edge (a, b) do
        if (number[b] == UNVISITED) then DFS(b)
    last[number[a]] := current - 1      // record preorder number of last descendant
```

Fig. 2. Depth-first search.

where we can enter a procedure; the distinguished node END has no outgoing edges and has incoming edges from every node with a return point. One can view START and END as standing in for everything external to the current procedure; an extra control-flow edge (START, END) represents the possibility that the procedure is not executed.

A node with multiple out-edges is a *branch*; such a node (with the exception of START) contains a final statement that is a conditional jump. A node with multiple in-edges, such as END, is a *merge*. For this article we are not concerned with labels on the different edges leaving a branch or entering a merge.

A *path* in G_{CF} from node a to node b is a sequence of zero or more edges in E_{CF} with the first having a as source, the last having b as destination. There is always a (zero-length) path from a node to itself.

2.2 Dominator Trees

The dominance relation records which nodes must be encountered en route to other nodes. A node a *dominates* a node b ($a \preceq b$) if all G_{CF} paths from START to b include a. When b has multiple dominators, the unique one that occurs after the other dominators on any path to b is b's *immediate dominator*.

Making the immediate dominator of each node its parent in a tree produces a dominator tree, such as that given rightmost in Figure 1. Dominance can be tested quickly using the preorder numbering trick described below for depth-first spanning trees. The dominator tree can be built in almost linear time [Lengauer and Tarjan 1979]

The dominator tree can be used directly to find reducible loops; we use this property in *fix_loops* later. For the analysis of irreducible loops, we rely on depth-first search.

2.3 Depth-First Search

A depth-first traversal of a graph visits all the nodes, marking them as they are visited. The next node visited is an unmarked neighbor of the most recently visited node with such a neighbor. If the traversal begins with START, and all other nodes are reachable, the edges followed define a depth-first spanning tree (DFST) of G_{CF}.

More precisely, depth-first search is implemented as in Figure 2. The nodes are ordered sequentially in the order that each is first visited during the search; the depth-first number of a node a (saved in number[a]) is equivalent to its preorder number on the DFST. By also saving the number of each node's last descendant

(last[number[a]]), we enable an efficient test for ancestry in the DFST. Given nodes a and b with number[a] = w and number[b] = v, we have

$$\text{isAncestor}(w, v) \equiv ((w \leq v) \textbf{ and } (v \leq \text{last}[w])).$$

The remaining algorithms refer to nodes by their preorder numbers except where unnumbered nodes have been inserted.

2.4 Loops

Intuitively, a loop is a chunk of code whose execution may repeat without the repetition of any surrounding code. More precise definitions rely on the notion of a *strongly connected region* (SCR): a nonempty set of nodes $S \subset N_{CF}$ for which, given any $q, r \in S$, there exists a path from q to r and from r to q [Aho et al. 1986]. An SCR to which no more nodes can be added while remaining strongly connected is a *maximal* SCR and fits the intuitive notion of an outermost loop (or of a single statement outside any loop).

Definition 1. An *outermost loop* is a maximal SCR with at least one internal edge.

Consider a path from START which visits exactly one node m in the SCR and ends there. The set of all nodes m in the SCR for which such a path exists is the set of *entries* for that SCR. In any particular depth-first search, the first node of the SCR to be visited (the one with the lowest preorder number) will be one of the entries. For an SCR that is also a loop, we define this first visited entry to be the *header* of the loop.

Definition 2. The loops nested inside a loop L with header h are the outermost loops with respect to the subgraph with node set $(L - h)$.

This definition is somewhat more arbitrary than that for outermost loops, but it has useful properties. We can represent the loop-nesting relation with a tree in which the subtree rooted by each header node contains exactly the nodes of the corresponding loop. For each node, the parent is the header of the smallest loop containing that node, or in the case of a header node for a loop L, the parent is the header of the smallest loop strictly containing L. We make the nesting tree rooted by giving START as the parent of outermost loop headers.

Definition 3. A *reducible* loop is a loop with a single entry; an *irreducible* loop is one with multiple entries.

Note that the unique entry of a reducible loop dominates all its other nodes, and will always be chosen as header regardless of the DFST used. For an irreducible loop, no one entry dominates the nodes of the loop. One of the entries will be chosen as header of the loop, but the choice will be arbitrary, based on the selection of DFST.

Figures 3 and 4 gives two possible depth-first spanning trees (with preorder numberings) and their corresponding loop-nesting trees for the example G_{CF}. In the loop tree, nodes with descendants are headers of loops, labeled **reducible** or **irreducible** accordingly. Nodes without descendants are either **nonheader** or **self**. **Self** is the single-node special case of a reducible loop.

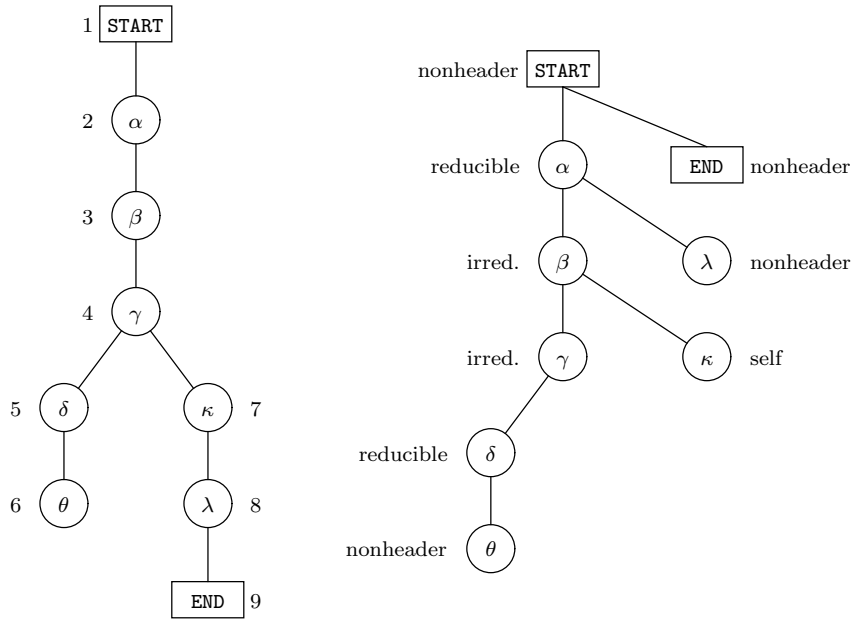

Fig. 3. DFST and loop tree for example.

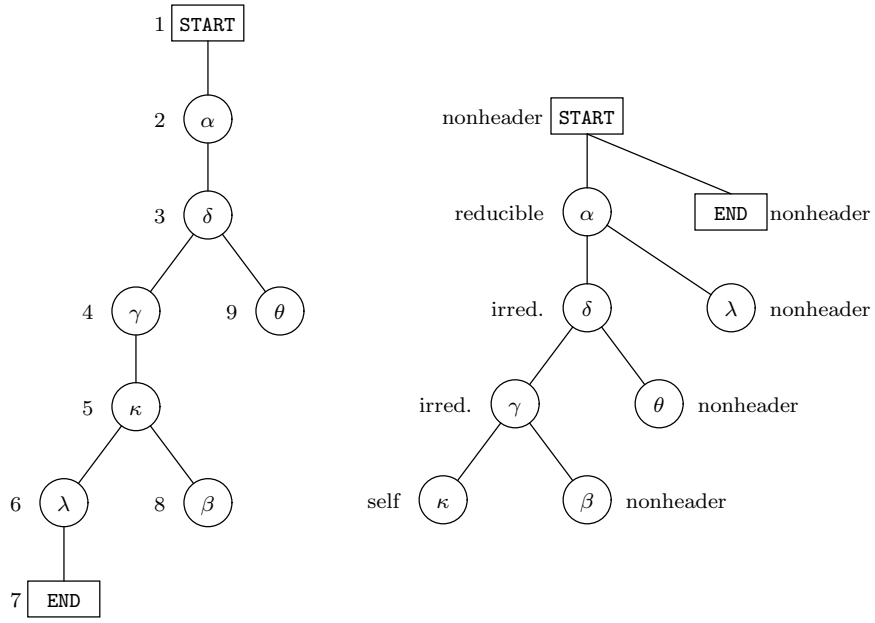

Fig. 4. Another DFST and loop tree for example.

Given a depth-first spanning tree of G_{CF}, any edge (v, w) where w is an ancestor of v in the spanning tree is called a *backedge*. The destination of a backedge is always a loop header, and each loop header will always have at least one incoming backedge.

When only reducible loops are present, our nested loops are equivalent to Tarjan intervals [Tarjan 1974]. The notion of nested reducible and irreducible loops presented here is a refinement of that presented in Havlak [1994] and Sloman et al. [1994].

2.5 UNION-FIND

Both Tarjan's method for testing reducibility and our procedure *analyze_loops* for building arbitrary loop-nesting trees make use of an algorithm for performing a sequence of UNION and FIND operations shown by Tarjan to take almost linear time [Tarjan 1983]. We omit code for the operations, defining only their behavior as used in this article.

UNION and FIND are employed on disjoint sets from a universe U of elements m_i, each identified by an integer $i \in [1, N]$. For our algorithms $U = N_{CF}$, and i is the DFST preorder number of node m_i. Each set $s_i \subseteq N_{CF}$ is also designated by an integer $i \in [1, N]$. Initially, $s_i = \{m_i\}$.

UNION operates to combine the sets, and FIND gives the set currently containing an element, taking all previous UNIONs into account. In our usage, UNION(x, y) assigns s_y equal to $s_x \cup s_y$ and then makes s_x empty. FIND(x) gives y such that $m_x \in s_y$.

3. BUILDING THE LOOP-NESTING TREE

Figure 5 gives our method for computing the headers of reducible and irreducible loops. The main structure of the procedure *analyze_loops* is that of Tarjan's method for testing reducibility, with one major difference. Tarjan's method quits on finding an irreducible loop, but ours marks the loop and continues [Tarjan 1974]. For reducible graphs, the methods are essentially the same.

We have kept fairly close to Tarjan's formatting for ease of comparison. The key variables (e.g., w, y', P) have the same names, and the letter labels on statements correspond to the same labels in Tarjan's formulation. Among the modest changes are specifying START as the header of outermost loop headers (so the loop-nesting tree will be rooted) and moving the UNION operation for the sake of clarity.

In step (a) we first number the nodes of G_{CF} in preorder using depth-first search and save the preorder number of each node's last descendant in the DFST. For the example program in Figure 1, this could produce either of the DFST's shown on the left of Figures 3 and 4 — let us proceed using only the DFST from Figure 3.

In Tarjan's presentation, step (b) builds lists of backedges and other edges into each node. A backedge comes from a descendant in the DFST and a nonbackedge from a nondescendant. It is sufficient to save only the sources of backedges and other edges separately, which we do in backPreds and nonBackPreds. The backedges, according to Figure 3, are $\{(\kappa, \kappa), (\theta, \delta), (\delta, \gamma), (\kappa, \beta), (\lambda, \alpha)\}$. We have also added the initialization of the node types to **nonheader**. The type field for nodes records distinctions that were implicit in Tarjan's work (nonheader vs. self) or absent (reducible vs. irreducible).

```
procedure analyze_loops(G_CF, START)
a:  number vertices of G_CF using depth-first search from START,
        numbering in preorder from 1 to |N_CF| and saving last[*]
b:  for w := 1 to |N_CF| do    // make lists of predecessors (backedge and other)
        nonBackPreds[w] := backPreds[w] := ∅
        header[w]       := 1       // default "header" is START
        type[w]         := nonheader
        foreach edge (v, w) entering w do
            if isAncestor(w, v) then add v to backPreds[w]
                                else  add v to nonBackPreds[w]
    header[1] := nil      // START is root of header tree
c:  for w := |N_CF| to 1 step −1 do
        P := ∅
d:      foreach node v ∈ backPreds[w] do
            if (v ≠ w) then add FIND(v) to P
                       else type[w] := self
        worklist := P
        if (P ≠ ∅) then type[w] := reducible
        while (worklist ≠ ∅) do
            select a node x ∈ worklist and delete it from worklist
e:          foreach node y ∈ nonBackPreds[x] do
                y' := FIND(y)
                if not(isAncestor(w, y')) then
                    type[w] := irreducible
                    add y' to nonBackPreds[w]
                else if (y' ∉ P) and (y' ≠ w) then
                        add y' to P and to worklist
        foreach node x ∈ P do
            header[x] := w
            UNION(x, w)      // merge x's set into w's set
```

Fig. 5. Finding headers of reducible and irreducible loops.

The outer loop at c is essentially unchanged; it does nothing interesting except for those nodes (the loop headers) which are the destinations of backedges. For a header node w, we chase backward from the sources of backedges adding nodes to the set P, representing the body of the loop headed by w. By running through the nodes in reverse of the DFST preorder, we ensure that each inner loop header will be processed before the headers for surrounding loops.

UNION and FIND play their important roles here. FIND gives representative nodes to be placed in P. At first, each node is its own representative. UNION merges the nodes of P into a set represented by w, the loop header. Then FIND(v) for any v will return the outermost loop header w whose set v has been UNION'ed into (so far). Each header represents all the nodes of its loop in the the P set for the next outer loop's header, so that the fine detail of each loop need only be examined once.

The loop at d starts off P with the backedge predecessors (actually, with their representatives from FIND). Every node added to P takes its turn in the worklist, then its nonbackedge predecessors are considered in loop e.

The changes to step e constitute the major difference from Tarjan's method. If, in chasing upward from the sources of w's backedges, we encounter a node y' that is

not a descendant of w, we then mark w as the header of an irreducible loop. Such a y' represents a path into w's loop that avoids w, implying another entry. (Note that we must add each such y' to the nonBackPreds set for w to ensure detection of surrounding irreducible loops entered through y'.) This is the point where Tarjan's method quits, reporting irreducibility.

Using the example of Figure 1 and its DFST in Figure 3, consider the processing when w is δ, γ, and β. In processing δ, the only backedge is from θ; and FIND(θ) yields θ; so the set P becomes $\{\theta\}$. The only nonbackedge predecessor of θ is δ, so we stop adding to P. We set header[θ] to δ.

When w equals γ, the only backedge is from δ, and FIND(δ) yields δ. We put δ in P and the worklist. Examining δ's nonbackedge predecessors in the **foreach** at e, we find α not a descendant of γ, meaning that from α there is another entry into γ's loop (through δ). We mark γ as irreducible and add α to γ's nonBackPreds set. We later set header[δ] to γ.

Now when we process w equal to β there is a backedge from κ. This causes no trouble, and its nonBackPred γ goes into the worklist. Having saved α as an extra nonBackPred of γ, we find that β is also not an ancestor of α and must be marked as irreducible.

When our algorithm completes, the header fields define the loop-nesting tree in Figure 1: header[x] becomes the parent of x in the tree. Note that Tarjan's method will provide this information only if every loop in the procedure is reducible; otherwise it will have quit with the terse answer of "not reducible."

4. MAXIMIZING REDUCIBLE LOOPS

Our goal in this article is to identify the reducible loops in irreducible graphs. Unfortunately, *analyze_loops* may miss some reducible loops, depending on the choice of DFST.

Figures 3 and 4 illustrate the problem. Node δ is one of the entries to an irreducible loop and may or may not be chosen as the header, depending on the DFST used. But δ may also be discovered as the header of the reducible loop $\{\delta, \theta\}$. Our definitions do not allow loops to share the same header, so the reducible loop will be discovered only if a different header is picked for the irreducible loop.

This problem cannot be fixed just by careful choice of DFST. One could construct a graph where every entry to an irreducible loop could also be the header of the reducible loop. Our definitions specify a different header node for each loop, and there are not enough to go around. Our solution is to insert empty nodes into G_{CF} so that each reducible loop (that we want to discover) will have its own header (so that it will be discovered).

Consider the set of backedges for which the destination dominates the source. These *reducible backedges* can be enumerated using the (unique) dominator tree for the procedure and do not depend on the choice of DFST. In our example G_{CF}, only the edges (λ, α), (θ, δ), and (κ, κ) are reducible backedges.

The other, *irreducible* backedges depend wholly on the DFST chosen. They are (δ, γ) and (κ, β) using the DFST in Figure 3, or (γ, δ) and (β, γ) using the DFST in Figure 4.

There can be no reducible loop without at least one reducible backedge. If we guarantee that every reducible backedge goes to the header of a reducible loop,

```
procedure fix_loops(G_CF, START)
    number vertices of G_CF using depth-first search from START,
        numbering in preorder from 1 to |N_CF| and saving last[*]
    for w := 1 to |N_CF| do
        redBackIn[w] := otherIn[w] := ∅
        foreach edge (v, w) entering w do
            if (w ⪯ v) then add (v, w) to redBackIn[w]
                       else add (v, w) to otherIn[w]
        if (redBackIn[w] ≠ ∅) and (|otherIn[w]| > 1) then
            insert new node w' into G_CF immediately before w, unnumbered
            create new edge (w', w)
            foreach edge (v, w) ∈ otherIn[w] do
                delete (v, w) and create otherwise identical edge (v, w')
```

Fig. 6. Splitting reducible loop headers from irreducible loops.

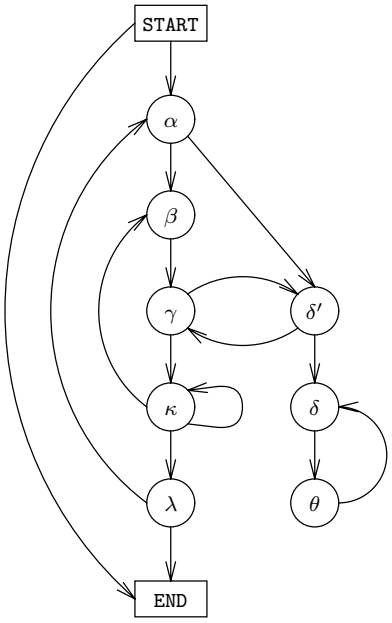

Fig. 7. G_{CF} for example with inserted nodes.

then we will in a sense have maximized the reducible loops. To do this, we must guarantee that no reducible backedge will share a destination with an irreducible backedge (for any DFST).

Procedure *fix_loops* in Figure 6 modifies G_{CF} so that any node with incoming reducible backedges has exactly one other incoming edge. Since every reachable node must have a tree edge in the DFST (the edge along which the depth-first search reached the node), there are no other edges left to be irreducible backedges.

The result of applying *fix_loops* to our example G_{CF} is given in Figure 7. Node δ now has one reducible backedge, from θ, and one other edge, from the new node δ'. All other edges into δ have been moved to δ'. Now δ' stands in as an entry (and possible header) to the irreducible loop.

5. RELATED WORK

Compiler researchers seem unified in identifying outermost loops with maximal strongly connected regions. There is less agreement on how to analyze nested loop structure.

Reducible loops with distinct headers have an obvious nesting structure; the only question is whether to recognize different loops with a shared header. Some formulations define one *natural loop* per reducible backedge [Aho et al. 1986]. A natural loop with its own header meets our definition of reducible loop. When natural loops share headers, they may or may not be properly nested — and interpreting them as nested loops may not be consistent with the programmer's intent. Avoiding such issues by treating natural loops that share a header as a single loop yields the same results as our definition of reducible loop (once *fix_loops* has been applied).

Researchers in interval analysis handle irreducible loops by identifying single-entry regions that enclose the loop. Schwartz and Sharir [1979] take the smallest reducible loop surrounding an irreducible loop as its interval. Burke [1990] takes the smallest single-entry connected region surrounding an irreducible loop as its interval — connected, but not strongly connected, so it is not necessarily a loop. The former has the certainty and the latter the possibility of defining the same interval for a reducible loop and an inner irreducible loop.

Sloman et al. [1994] provide definitions of nested loops that are similar to ours. They do not require that reducible and irreducible loops have different headers, so they do not require preprocessing as in *fix_loops*. They also combine irreducible loops whose contents have the same last common dominator, so that the set of irreducible loops found is invariant with the choice of DFST.

6. CONCLUSIONS

We have presented methods to find a maximal number of reducible loops in arbitrary control-flow graphs. Discovering more reducible loops would require transformations based on additional program knowledge outside the flowgraph.

The only nonlinear-time complexities are in UNION-FIND and the construction of the dominator tree, both of which are almost linear [Tarjan 1983; Lengauer and Tarjan 1979]

We expect these methods to be useful in loop-optimizing compilers which must handle arbitrary control-flow. Reducible loops can be recognized in any context and subjected to extensive dependence analysis and restructuring.

ACKNOWLEDGEMENTS

Michael Wolfe and Eric Stoltz pointed out the necessity of adjusting the predecessors set of a header discovered to be irreducible and helped with many other comments. To them and to my colleagues on the ParaScope and D System projects at Rice and the CHAOS project at Maryland I am deeply thankful.

REFERENCES

Aho, A. V., Sethi, R. I., and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, Reading, Mass.

Burke, M. 1990. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Trans. Program. Lang. Syst. 12,* 3 (July), 341–395.

Havlak, P. 1994. Interprocedural symbolic analysis. Ph.D. thesis, Tech. Rep. CS-TR94-228, Dept. of Computer Science, Rice Univ., Houston, Tex.

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

Schwartz, J. and Sharir, M. 1979. A design for optimizations of the bitvectoring class. Tech. rep., Courant Inst., New York Univ., New York. September.

Sloman, B., Lake, T., and William, S. 1994. Identifying loops in flowgraphs. Tech. Rep. SEG/GN/94/1, Univ. of Reading, Reading, UK.

Tarjan, R. E. 1974. Testing flow graph reducibility. *J. Comput. Syst. Sci. 9*, 355–365.

Tarjan, R. E. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Pa.

Received November 1995; revised August 1996; accepted October 1996