Flow Graph Anomalies: What's in a Loop?

Michael Wolfe

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 92-012

February 1991

Flow Graph Anomalies: What's in a Loop?

Michael Wolfe*

February 28, 1991

Abstract

One of the most basic analysis techniques in compilers is the identification and optimization of loops. Three classical loop identification techniques are to find back edges in the dominator tree, to use interval analysis, and to use T1-T2 analysis. Recent work has developed a new compiler intermediate form which uses the control dependence graph instead of the control flow graph to represent control relationships between program elements. The control dependence graph can also be used to directly identify loops in a program. Here we show that in some unstructured programs, the loop nesting identified by the control dependence graph is different than that identified by the classical methods; that is, the control dependence graph will say that there are two loops, L1 and L2, and that L1 is nested within L2, while the other techniques will say that L2 is nested within L1. This difference raises the question of which is the "right" definition, or if in fact the answer for such a program has a meaningful definition.

1 Introduction

Loop identification is one of the most basic compiler analysis methods used for optimization. Some language processors can survive using only syntactic loop identification, that is, only identifying the loops explicitly identified in the source program as loops (such as WHILE loops, REPEAT loops and FOR loops). This type of analysis would completely miss loops arising from explicit GOTO statements, such as:

```
l1:  A(i) = B(i) + 1
     i = i + 1
     if( i < 10 ) goto l1
```

Most compiler researchers (and indeed most commercial compiler implementations) use a flow graph representation of a program; for our purposes a flow graph is defined as follows:

*Supported by NSF Grant CCR-8906909 and by DARPA Grant MDA972-88-J-1004.

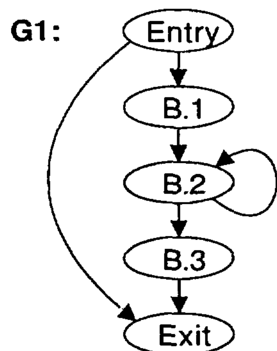

Figure 1: Flow graph for a simple loop.

A flow graph is the tuple $G = \langle V, E, Entry, Exit\rangle$, where V is a set of vertices corresponding to basic blocks, (alternatively, the vertices in V may correspond to statements or operations), E is a set of directed edges $E = \{ (m,n) \mid m, n \in V$ and m is a flow predecessor of $n \}$, $Entry \in V$ and $Exit \in V$ are distinguished vertices. We will write $m \rightarrow n$ to mean there is an edge $(m,n) \in E$. A path in a flow graph from vertex m to vertex n is a length k sequence of vertices $(v_0, v_1, v_2, \ldots, v_k)$, with $v_i \in V\ \forall i, 0 \leq i \leq k$ and $v_0 = m$ and $v_k = n$, such that $v_0 \rightarrow v_1$, $v_1 \rightarrow v_2$, $\ldots$, $v_{k-1} \rightarrow v_k$. By convention, we say there is always a length-0 path from a vertex to itself. G is connected with a path from *Entry* to any vertex $n \in V$ in G and with a path from any vertex $n \in V$ to *Exit* in G.

The edge from *Entry* to *Exit* is included to allow the algorithms for finding dominators, post-dominators and dominance frontiers work more simply Usually flow graphs are represented pictorially; Figure 1 shows the flow graph for the following program:

```
L.1:  A = ...
L.2:  B = ...
      if( B > 0 ) goto L.2
L.3:  C = ...
```

where block B.i in the figure corresponds to the basic block beginning at label L.i. We might be tempted to say that a loop in the program is a strongly connected region of the flow graph; that definition does not distinguish nested loops. Instead, the flow analysis techniques use a more precise definition of loop, as described in the following sections.

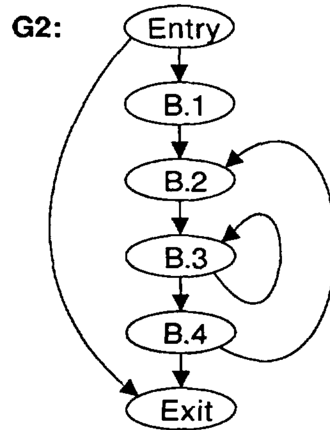

Figure 2: Flow graph for a doubly nested loop.

2 Five Sample Flow Graphs

This paper compares the results of different control flow analysis techniques on certain flow graphs. This section gives several interesting flow graphs and predicts the questions that might be asked about them. The following sections will show how each analysis technique defines the loops in each flow graph. The first flow graph is the simple loop shown in Figure 1, which we include for completeness; each of the analysis techniques will determine that such a flow graph includes a single loop comprising {B.2}.

The second flow graph is a simple doubly nested loop, shown in Figure 2. This example is given to show how the different techniques handle nested loops. All of the techniques will find the two nested loops, with the inner loop comprising {B.3}, and the outer loop comprising {B.2, B.3, B.4}.

The third flow graph is shown in Figure 3. This is an unstructured flow graph; there seems to be two loops in this graph, corresponding to the two back edges, but since they are improperly nested it is not immediately clear that one of the loops contains the other loop. Each of the flow analysis techniques does indeed find two loops in this flow graph, though the control dependence graph defines the inner and outer relationship differently than the classical techniques.

The fourth flow graph is shown in Figure 4. Again there seems to be two loops in this flow graph, with the loop {B.1, B.2, B.3} contained within {B.1, B.2, B.3, B.4}. We will find that the control dependence graph identifies two distinct loops, as we expect; interval analysis will identify only a single loop, however, while dominator tree back edge detection and T1-T2 analysis are ambiguous in this case.

The final two flow graphs studied here are shown in Figure 5. The first flow

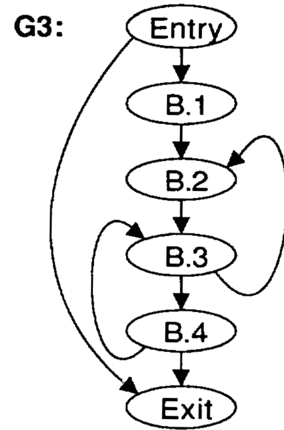

Figure 3: Flow graph for an unstructured loop.

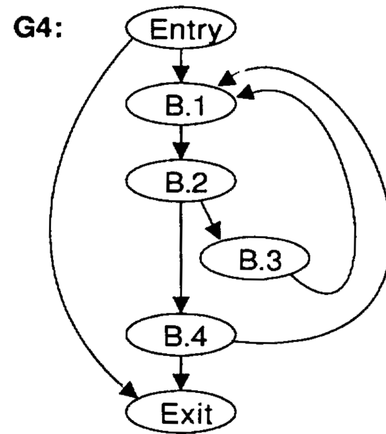

Figure 4: Flow graph for loops with common header.

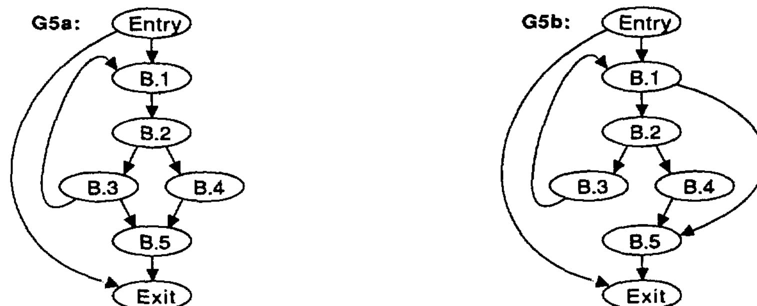

Figure 5: Flow graphs for loops with internal exit.

graph corresponds to a REPEAT UNTIL loop, and the second graph corresponds to a WHILE loop; each loop has an internal exit branch, with some code executed on the exit path (block B.4). The question here is whether block B.4 is contained within the loop or not. We will see that the control dependence graph determines that B.4 is not part of the loop, even though its execution depends on the loop.

3 Dominator Back Edge Method for Loop Detection

One of the classical methods to find loops in a flow graph and determine what vertices belong to the loop is to find back edges in the dominator tree. The dominator relationship is defined as follows:

> Given a flow graph G, a vertex m dominates a vertex n, written m DOM n, iff every path in G from *Entry* to n contains m.

By convention, the dominator relationship is reflexive. Let DOM(n) be the set of nodes $\{m \mid m \text{ DOM } n\}$. The dominator relationship for a flow graph can be represented by a tree with vertices from V, rooted at *Entry*, such that $m \in \text{DOM}(n)$ iff $m = n$ or m is an ancestor of n in the dominator tree [4]. The dominator trees for our sample flow graphs are given in Figures 6-9. Given the dominator tree, a back edge is an edge $n \rightarrow m \in E$ such that m DOM n. In other words, a back edge is one that goes from a vertex to one of its ancestors in the dominator tree. The back edges are listed in the following table for each flow graph:

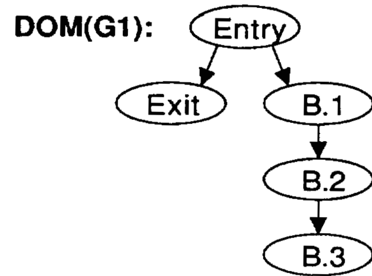

Figure 6: Dominator tree for the simple loop G1.

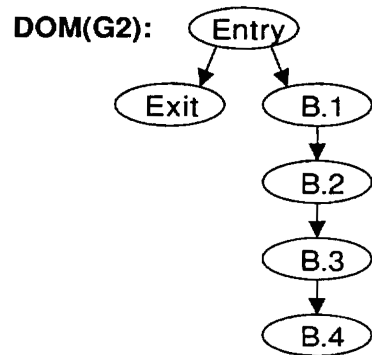

Figure 7: Dominator tree for the doubly nested loops G2 and G3.

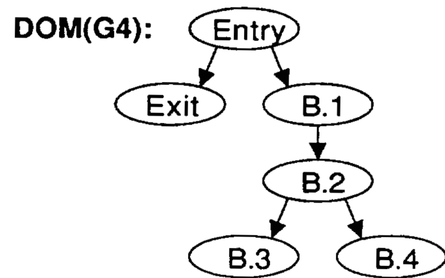

Figure 8: Dominator tree for loops with common header in G4.

Figure 9: Dominator trees for loops with internal exit in G5a and G5b.

G1	B.2 → B.2	
G2	B.4 → B.2	B.3 → B.3
G3	B.4 → B.2	B.3 → B.1
G4	B.4 → B.2	B.3 → B.2
G5a	B.3 → B.1	
G5b	B.3 → B.1	

Back edge detection would try to define a loop for every back edge. The body of the loop contains the loop header, defined as the target of the back edge, and all vertices that can reach the back edge without going through the header. The loop headers and loop bodies for all graph except G4, as determined by back edge detection, are given in the table below:

	back edge	header	body
G1	B.2 → B.2	B.2	{B.2}
G2	B.3 → B.3	B.3	{B.3}
	B.4 → B.2	B.2	{B.2, B.3, B.4}
G3	B.4 → B.2	B.2	{B.2, B.3, B.4}
	B.3 → B.1	B.1	{B.1, B.2, B.3, B.4}
G5a	B.3 → B.1	B.1	{B.1, B.2, B.3}
G5b	B.3 → B.1	B.1	{B.1, B.2, B.3}

When two loops have different headers, the two loops are either disjoint or one is contained entirely within the other. In G2 we see that the loop headed by B.3 is contained in (nested in) the outer loop headed by B.2. In G3, we see that the loop headed by B.2 is nested in the loop headed by B.1; this is because the outer loop back edge can be reached by the path B.1 → B.2 → B.3 → B.4 → B.2 → B.3 → B.1.

One problem with this technique is the relative nesting levels of loops derived from back edges with the same loop header is not always clear. Graph G4 has two back edges with the same loop header, B.2. Taking the back edges one at a time, the two loops found are:

	back edge	header	body
G4	B.3 → B.1	B.1	{B.1, B.2, B.3}
	B.4 → B.1	B.1	{B.1, B.2, B.4}

neither of which is contained totally within the other. We might look only at the edge B.4 → B.1, allowing the B.3 → B.1 edge as a path in that loop. Then this would generate the desired inner and outer loops:

	back edge	header	body
G4	B.3 → B.1	B.1	{B.1, B.2, B.3}
	B.4 → B.1	B.1	{B.1, B.2, B.3, B.4}

However, an equally valid attempt would be to look only at the edge B.3 → B.1, allowing the B.4 → B.1 edge as a path in that loop; this would generate the other inner/outer relationships the other way:

	back edge	header	body
G4	$B.3 \rightarrow B.1$	B.1	{B.1, B.2, B.3, B.4}
	$B.4 \rightarrow B.1$	B.1	{B.1, B.2, B.4}

One method used to disambiguate this problem is to treat this flow graph as a single loop with two back edges [1].

4 Interval Analysis for Loop Detection

Another classical method to find loops is to use interval analysis. An interval with header vertex n, denoted $\mathrm{I}(n)$, is the set of vertices

$$\mathrm{I}(n) = \{\ m \mid m = n \text{ or } \mathrm{pred}(m) \subseteq \mathrm{I}(n)\ \},$$

where pred(m) is the set of vertices

$$\mathrm{pred}(m) = \{\ p \mid p \rightarrow m \in E\ \}.$$

A flow graph can be partitioned into disjoint intervals; the interval partition then induces a new flow graph, called the interval graph or $\mathrm{I}(G)$, whose vertices are the interval partitions of G, with entry vertex $\mathrm{I}(\mathit{Entry})$, and with edges $\mathrm{I}(m) \rightarrow \mathrm{I}(n)$ where $m \rightarrow n \in E$ and $\mathrm{I}(m) \neq \mathrm{I}(n)$. The interval algorithm can then be run again on the interval graph, and so on, generating an interval graph sequence. Intuitively, an interval is a loop with some dangling edges. Outer loops will appear later in the interval graph sequence than inner loops. If the interval graph sequence terminates with a single node, then the flow graph is said to be reducible.

The properties of an interval include that each interval has a unique header vertex, and the header dominates all vertices in the interval. If there is an edge $x \rightarrow h$ from some vertex x to the header h, then that interval includes a loop. The interval graph sequence for our sample flow graphs are given in the table below; those intervals that correspond to loops are noted with an asterisk. Note that the problem of multiple back edges ending on the same header is well defined with interval analysis.

G1	{*Entry*,B.1}	{B.2,B.3}*	{*Exit*}		
G2	{*Entry*,B.1}	{B.2}	{B.3}*	{B.4}	{*Exit*}
I(G2)	{*Entry*,B.1}	{B.2,B.3,B.4}*	{*Exit*}		
G3	{*Entry*}	{B.1}	{B.2,B.3,B.4}*	{*Exit*}	
I(G3)	{*Entry*}	{B.1,B.2,B.3,B.4}*	{*Exit*}		
G4	{*Entry*}	{B.1,B.2,B.3,B.4}*	{*Exit*}		
G5a	{*Entry*}	{B.1,B.2,B.3,B.4,B.5}*	{*Exit*}		
G5b	{*Entry*}	{B.1,B.2,B.3,B.4,B.5}*	{*Exit*}		

Interval analysis does not explicitly define the body of a loop, so similar techniques as used in dominator back edge detection could be used. Interval analysis finds the same loops as back edge detection, with the same inner/outer relationship (see G3), except for graph G4. With back edge detection, this case (multiple back edges to the same loop header) was ambiguous; with interval analysis, this is unambiguously defined as a single interval, and thus is treated as a single loop.

4.1 T1-T2 Analysis

T1-T2 analysis is another method to "parse" a flow graph; it reduces a flow graph by the repeated application of the two transformations:

T1 if n is a node with a loop edge $n \rightarrow n$, delete that edge.

T2 if there is a node n with a unique predecessor m, delete the edge $m \rightarrow n$ and the node n, merge the successors of the two nodes (making all successors of n be successors of m).

Intuitively, inner loops will be reduced by the T1 transformation before outer loops. T1-T2 analysis gives the same results as interval analysis except for graph G4. In that case, depending on the order of the transformations, the analysis might find two loops nested one way, two loops nested the other way, or only one loop, as shown in Figures 10-12.

5 Control Dependence for Loop Detection

A relatively new method to represent the conditions controlling execution of vertices in the program graph is the control dependence relationship [3]. To understand control dependence we must first define the postdominator relation.

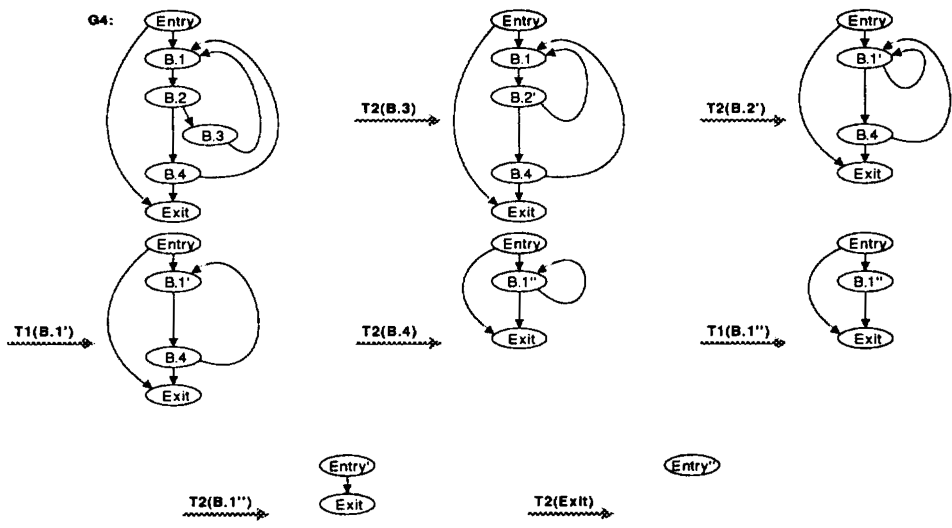

Figure 10: T1-T2 analysis of graph G4 finding two nested loops.

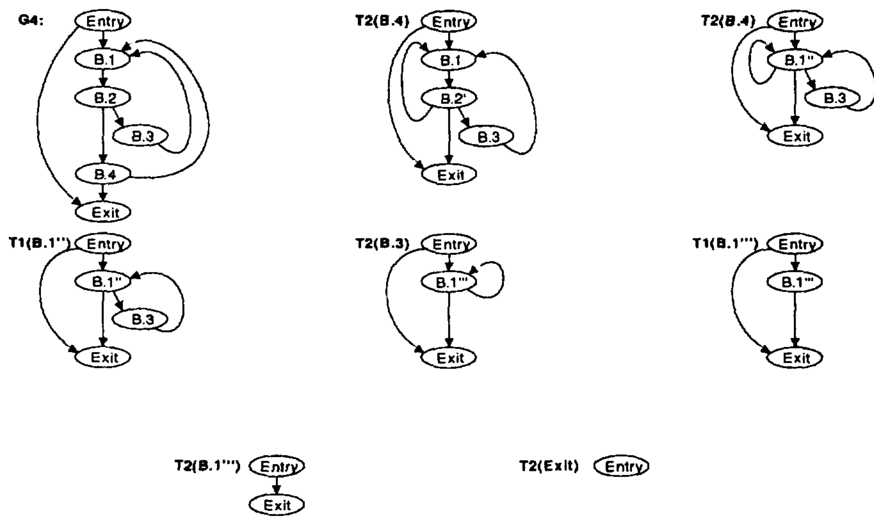

Figure 11: T1-T2 analysis of graph G4 finding two loops nested other way.

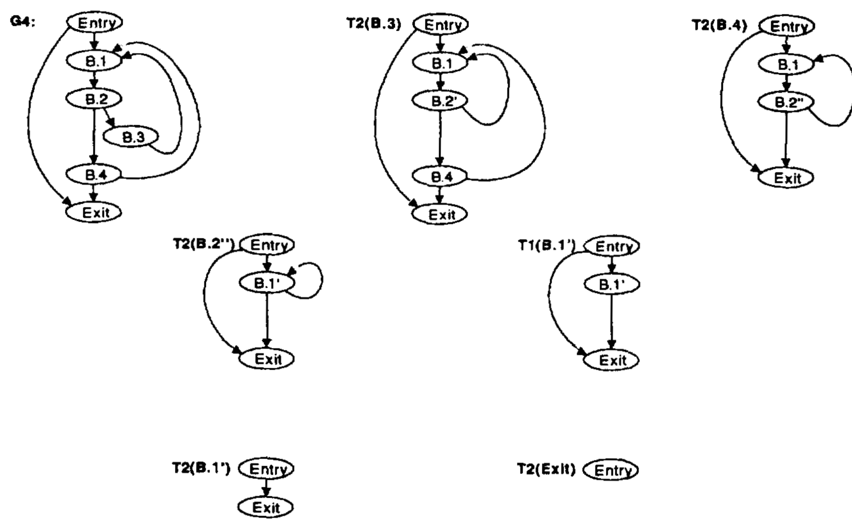

Figure 12: T1-T2 analysis of graph G4 finding a single loop.

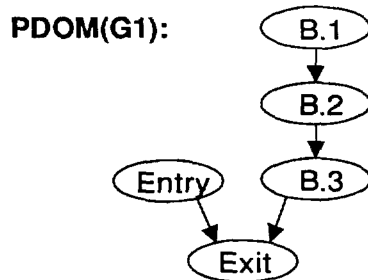

Figure 13: Post-dominator tree for the simple loop G1.

We say that a vertex n postdominates a version m, written n PDOM m, if n appears on every path from m to *Exit*. Like the dominator relationship, the postdominator relationship is reflexive and transitive, and can be represented in a postdominator tree. Cytron et al [2] explain that the postdominator relationship is the same as the dominator relationship of the reverse control flow graph (reversing each edge in the control flow graph), and they show an efficient method to find control dependences from the postdominator tree.

A vertex n is control dependent on another vertex m, if there is a length $k > 0$ path from a m to n such that n postdominates every vertex after m on the path, and either $n = m$ or n does not postdominate m. Note that a vertex can be control dependent on itself. The control dependence relationship can be represented by a directed graph, called the control dependence graph. The only control dependence predecessors are those vertices that conditionally choose one of two (or more) successor paths in the program. Each arc in the graph is labelled to indicate whether the successor vertex is on the True or False path (integer labels would be used if there were more than two successor paths). The post-dominator trees for the sample flow graphs are given in Figures 13-16.

A cycle in the control dependence graph corresponds to a loop in the program. The control dependence cycle corresponds to the condition that a loop exit can decide to exit the loop or continue execution execution of the loop only if the previous loop exit (perhaps from an earlier iteration of the loop) has not already decided to exit the loop. The body of a loop is control dependent upon the dependence cycle. Nested loops appear as disjoint cycles, with the inner loop cycle control dependent upon the outer loop cycle. The control dependence graphs for the first two simple example flow graphs are shown in Figures

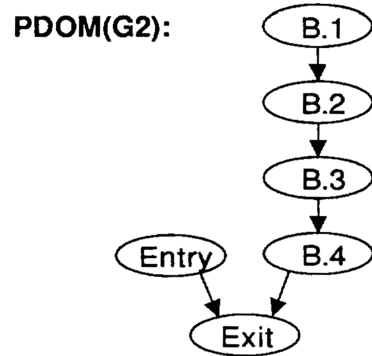

Figure 14: Post-dominator tree for the doubly nested loops G2 and G3.

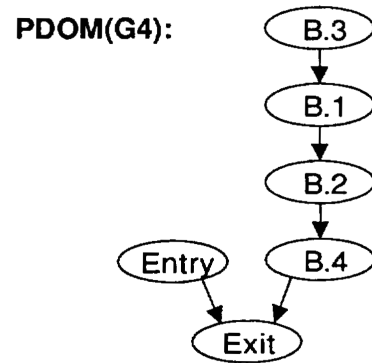

Figure 15: Post-dominator tree for loops with common header in G4.

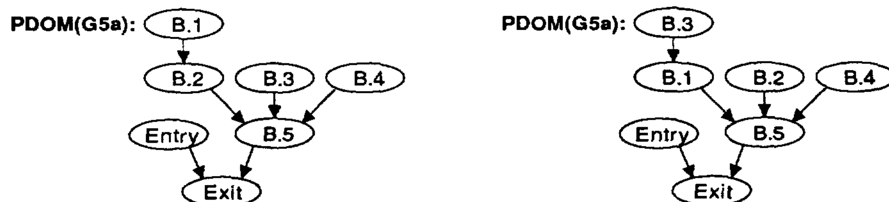

Figure 16: Post-dominator trees for loops with internal exit in G5a and G5b.

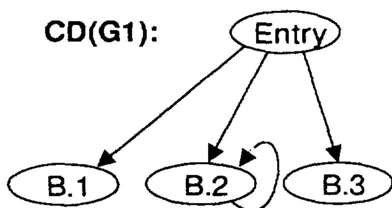

Figure 17: Control dependence graph for the simple loop G1.

14 and 15. The labels on control dependence arcs are omitted if the label is "T", to reduce clutter. In these two cases we get exactly the loops we expect; in G2 we see the two loops, with loop B.3 and vertex B.2 nested within the loop controlled by B.4.

The control dependence graph for G3, however, gives us our first surprise. This graph is shown in Figure 16. Here we see the two loops we expected, controlled by B.4 and B.3. However, the outer loop is controlled by B.4; the inner loop is controlled by B.3 and contains B.1 and B.2. This is exactly opposite of the inner/outer loop relationship found by the classical methods; this difference raises the question of whether the control dependence definition might be incorrect.

Flow graph G4 could be defined as having one or two loops by classical methods; the control dependence graph (in Figure 17) unambiguously shows

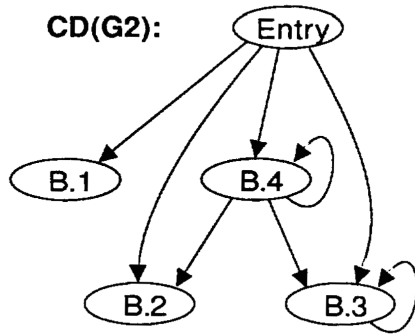

Figure 18: Control dependence graph for the doubly nested loops in G2.

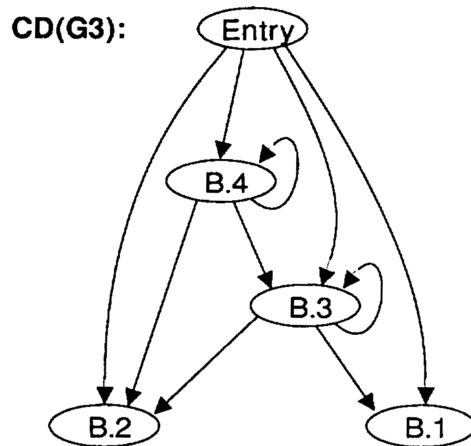

Figure 19: Control dependence graph for the improperly nested loops in G3.

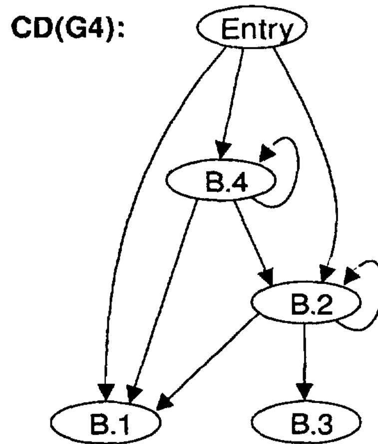

Figure 20: Control dependence graph for loops with common header in G4.

two nested loops, exactly as our intuition would expect. The inner loops is controlled by B.2, and contains B.3 and B.1; note that B.3 is control dependent only on B.2. This example shows a strength of the control dependence graph; it is precise even when loops share headers, when classical methods can be ambiguous. At this point we should examine our intuitive definitions of inner and outer loops. Given two nested loops, we expect that the inner loop will iterate several times for each iteration of the outer loop. In general, depending on the iteration condition, the inner loop may iteration only once or even zero times for some or all iterations of the outer loop; nonetheless, when the inner loop exits, it gets another chance to execute when the outer loop iterates. Thus, iteration of the inner loop "depends" on the outer loop iterating. The control dependence graph appears to capture this intuitive definition more precisely.

The question about flow graphs G5a and G5b is whether vertex B.4 belongs to the loop. Note that syntactically, that vertex may come from a basic block that is lexically scoped within the loop or not. In both graphs, shown in Figure 18, vertex B.4 does depend on the control dependence cycle defining the loop (note that the control dependence cycle comprises two vertices in both graphs). In both cases, however, B.4 depends on B.2 with a "False" label (the exit label of the loop). Depending on the application, we could define B.4 to be part of the loop, due to the dependence on the loop dependence cycle, or we could define B.4 to be outside of the loop, since it depends only on an exit label of the loop dependence cycle. We believe the proper definition is the latter; the code could be treated as a loop with a CASE or SWITCH statement choosing which exit code to execute when the loop exits. This definition can be easily supported by the control dependence graph.

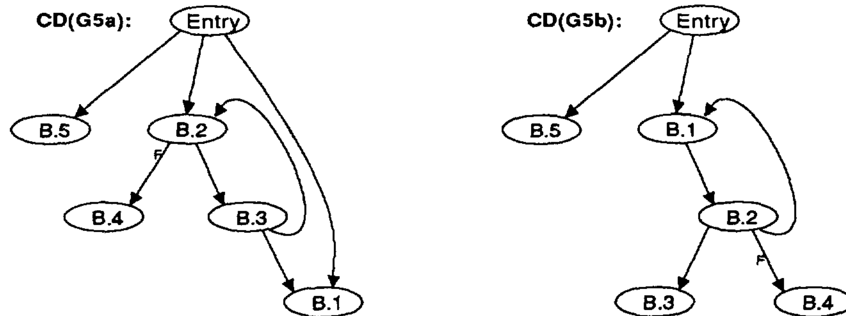

Figure 21: Control dependence graphs for loops with internal exit in G5a and G5b.

6 Summary

The control dependence graph, a new formal representation of the conditions that control execution of statements and blocks in a program, has been used as a basis of many scalar optimizations as well as the discovery of parallelism in sequential programs. One point to consider when adopting a new formalism is how it defines standard control constructs compared to other methods. We have compared how the control dependence graph defines loops, especially how it defines inner and outer loops, compared to the classical methods of back edge detection and interval analysis. We found that in one flow graph, the control dependence graph defines inner and outer loops unambiguously when classical methods are somewhat ambiguous; in another flow graph, the control dependence graph defines inner and outer loops differently from the classical methods. These differences arise in unstructured cases where our intuitive definition of inner and outer loop is not so easy to apply. It is disconcerting that the control dependence definition of loop can be different from the classical definition of loop. It raises the question of the validity of the control dependence graph, or of the validity of the definition of a "natural loop" of a program. Nonetheless, we prefer the definition from the control dependence graph over that of classical methods.

Experiments are needed (and are underway) to determine whether the differences appear in actual programs. This likely will depend on the programming language and application domain; our expectations are that few programs will exhibit the flow graphs that cause the problems shown here. If this bears out, any loop detection algorithm should be considered sufficient.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques,*

and Tools. Addison-Wesley, Reading, MA, 1976.

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conf. Record 16th Annual ACM Symp. on Principles of Programming Languages*, pages 25–35, Austin, TX, January 1989.

[3] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[4] Matthew S. Hecht. *Flow Analysis of Computer Programs.* North Holland, New York, 1977.