THE USE OF CONTROL-FLOW AND CONTROL DEPENDENCE IN SOFTWARE TOOLS

by

THOMAS JAUDON BALL

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1993

ABSTRACT

Program development, debugging, and maintenance can be greatly improved by the use of software tools that provide information about program behavior. This thesis focuses on a number of useful software tools and shows how their efficiency, generality, and precision can be increased through the use of control-flow and control dependence analysis. We consider two classes of tools: *execution measurement* tools, which collect information about a particular program execution; and *program analysis* tools, which provide information about potential program behavior by statically analyzing the program.

We consider three tools that measure aspects of a program's execution: profiling, tracing, and event counting tools. We describe algorithms for profiling and tracing programs that use a combination of control-flow analysis and program instrumentation to produce *exact* profiles and traces with low run-time overhead. Rather than record information at every point in a program, the algorithms record information at a subset of points that uniquely determines the information at unmeasured points. A unique part of our work is to classify various profiling problems, based on what is profiled and where profiling code is placed, and to compare the run-time costs of the various problems. Event counting is a special form of profiling that maintains an aggregate count of events in an execution.

We also consider three *semantics-based* software tools: slicing, differencing, and integration. A slicing tool computes a projection of a program that preserves the behavior of the original program at a particular statement. Such a tool is useful for debugging, since statements not in the projection have no effect on the statement of interest. Differencing compares the behavior of two programs, and integration merges variants of an original program so as to preserve the changed computations in each variant with respect to the original program. To date, no work has adequately addressed how to slice, difference, and integrate programs with complex control-flow (*i.e.*, programs containing unconditional jumps such as GOTOs). We show how to extend slicing,

differencing, and integration to languages with complex control-flow. Our results on control dependence are the basis for extending these tools to handle a larger class of languages.

ACKNOWLEDGEMENTS

First, I must thank Susan Horwitz, my advisor, for her guidance and support. Her enthusiasm, insight, and direction have been a great help and inspiration throughout my graduate years.

Collaborating with James Larus was also a key part of my graduate experience. Thanks to Jim for his part in developing the profiling/tracing tool *qpt*.

I would also like to thank the other members of my committee, Marvin Solomon, Thomas Reps, and Joel Robbin for their comments on my work.

Living and working in Madison has been a great pleasure, due in no small part to the many kind people in the department I have met along the way: Paul Adams, Samuel Bates, Dave Binkley, Edie Epstein, Lorenz Huelsbergen, Phil Pfeiffer, G. Ramalingam, Genevieve Rosay, Divesh Srivastava, and Wuu Yang. For the many jam sessions, thanks to Eric Bach and Joel Robbin. To those that I have not named here, thanks to you too.

I would not have made it here without my loving parents, the Drs. Charles and Eleanor Ball, who have always been a positive and motivating influence in my life.

A big thank you to my wife, Catherine Ramsey, for her love and companionship over the past four years, especially in this last difficult, crazy, and wonderful year of graduate school. Thanks also to David Ramsey Ball, whose cheerful disposition and amazing sleep schedule have helped to keep Catherine and me happy and alert in the last half year.

Chapter 1

INTRODUCTION

Program development, debugging, and maintenance can be greatly improved by the use of software tools that provide information about program behavior. This thesis focuses on a number of useful software tools and shows how their efficiency, generality, and precision can be increased through the use of control-flow [1] and control dependence analysis [17]. We consider two classes of tools: *execution measurement* tools, which dynamically collect information about a particular execution of a program; and *program analysis* tools, which provide information about potential program behavior by statically analyzing the program.

In the area of execution measurement, we consider three tools that provide information about the behavior of a program's execution: profiling, tracing, and event counting. A basic block or control-flow edge profiler records the number of times each basic block or control-flow transfer in a program occurs in an execution. Program profiles have many uses: they are used during program testing to determine which parts of a program have been exercised and, in program performance tuning, to point to "hot spots" in program execution, where improvements can be made [26]. Profile information also assists many compiler optimizations [19, 48, 53]. Tracing tools record the sequence of events in a program's execution [45]. Traces are often used for performance analysis, simulation, and debugging. Finally, event counting is a special form of profiling that maintains an aggregate count of the number of events that have occurred in a program's execution. Event counting is needed in applications that require efficient on-line querying of event information, which is not possible with basic block counters.

Other tools provide information about possible program behavior solely by the static analysis of programs. We consider three such *semantics-based* software tools: slicing, differencing, and integration tools. A slicing tool computes a projection (slice) of a program that preserves the behavior of the original program at a particular program statement [70]. Such a tool is clearly useful for debugging, since a slice ignores those parts of a program that cannot contribute to a particular computation. The operation of slicing is also important to differencing and integrating. A differencing tool compares two programs and determines those points at which the programs may compute different values [31]. An integration tool takes three programs as input, an original program and two variants, and determines whether the programs can be merged in such a way as to preserve the changed computations in each variant with respect to the original program [30]. A main application of program integration is reconciling divergent lines of program development in a multi-programmer project. Other uses are described by Horwitz and Reps [34]. In both differencing and integration, slicing is used to conservatively determine when two points in different programs have equivalent behavior. In integration, slicing is also used to extract relevant code from the input programs to form the merged program.

The first two contributions of this thesis are efficient algorithms for profiling and tracing programs. These algorithms use a combination of control-flow analysis and program instrumentation to produce *exact* profiles and traces with low run-time overhead. Sampling, another popular method, periodically examines the state of the program's execution but produces inexact results. Many instrumentation-based approaches to profiling and tracing incur high overhead because of the large amount of extra code introduced into the program. Rather than record information at every point in a program, the algorithms described here record information at a subset of points that uniquely determines the information at other points in the program. Furthermore, when possible, code is placed in areas of the program that execute less frequently, further decreasing the run-time overhead. Control flow analysis is used both to determine those places at which to record information and to generate full profile or trace information from the recorded information.

As described in Chapter 3, there has been considerable work on efficiently profiling and tracing programs. Three factors significantly distinguish our work from previous work. First, we consider both the theoretic and algorithmic underpinnings of program profiling and tracing. Second, unlike most previous work, we implemented the algorithms and experimented with different instrumentation strategies on a collection of real programs. This experience exposed deficiencies in previous algorithms and led to extensions that make these algorithms robust enough for practical use. Third, we implemented and compared several strategies for profiling and tracing. These approaches can be categorized as to whether they measure basic block or control-flow edge frequency, and whether they place instrumentation code in basic blocks or along control-flow edges. This categorization helps to relate the efficiency of various approaches. Through this categorization, we identified a new problem that has not been previously considered: basic block profiling with edge counters. We characterize this new problem and compare it to existing approaches.

The third contribution of this thesis is an algorithm for efficiently counting events in a program's execution, with support for on-line queries of the event count. We present a new method for efficiently counting and querying program events that uses program instrumentation. As with the profiling and tracing algorithms, this algorithm finds a subset of the points in a program to instrument while guaranteeing that accurate event counts can be obtained efficiently at every point in the execution.

Slicing, differencing, and integration tools have been developed for languages with scalar variables, structured control-flow, and multiple procedures [8, 30]. Related work in the area of alias and dependence analysis can aid in extending slicing to programs with arrays and pointers [7, 43, 56]. Other differencing and integration tools are not semantics-based. For example, the tools *diff* [35] and *diff3* are text-based and can be applied to any programs (or arbitrary text files, for that matter). However, no semantic guarantees can be made about the results of such tools (*diff3* may not even produce programs that are syntactically correct).

To date, no work has fully addressed the issues of slicing and semantics-based differencing, and integrating of programs in a language with more complex control-flow (*i.e.*, programs containing unconditional jumps such as **break**, **continue**, **goto**, etc.) In fact, some existing slicing algorithms compute semantically incorrect program projections in the presence of complex control-flow. Other algorithms that do handle **goto**s are overly conservative in nature, producing larger projections than necessary. Chapters 6 and 7 show how to extend slicing, differencing, and integration to languages with complex control-flow. Control dependence plays a key role in all three tools.

The main problem in slicing programs with complex control-flow is to identify when unconditional jumps are required in a slice in order to preserve a computation. Using the standard control-flow translation, existing slicing algorithms, which use backwards closure over data and control dependences (relations defined over the control-flow graph) to identify the statements of a slice, do not correctly identify when an unconditional jump is required. We show how an *augmented* control-flow translation enables these algorithms to compute correct slices. With the augmented translation, control dependences correctly identify when unconditional jumps are required in a slice.

We show how our results for slicing programs with complex control-flow allow us to compare the execution behaviors of statements in *different* programs. The ability to compare behaviors across programs leads to a differencing algorithm for programs with complex control-flow. The integration algorithm also makes use of this ability. This algorithm has three basic steps:

(1) Identify those statements whose computations in either one of the variants differ from the original program (*i.e.*, changed behaviors) and those statements whose computations are the same in all three programs (*i.e.*, preserved behaviors);

(2) Identify those slices of each program that should be incorporated in the merged program in order to replicate the changed and preserved behaviors;

(3) Combine the slices identified in step (2) into a merged program so that the behavior of each individual slice is correctly replicated. (In general, it may not be possible to construct such a program because the changed behaviors from the variants may interfere with one another.)

Our new slicing algorithm allows us to perform steps (1) and (2) for a language with complex control-flow. Step (3) presents an unusual problem: when merging the slices into one program, the statements must be ordered with respect to one another to ensure that the data dependences and control dependences from the slices are preserved in the merged program and that no new dependences are introduced (in order to guarantee that the behavior of the slices is correctly replicated). We refer to this process as *reconstitution*. We have shown that a reconstitution algorithm for a language with structured control-flow is correct [3]. The difficult part of this algorithm is to order the program statements to preserve data dependences. With the introduction of more complex control-flow into a language, the reconstitution process becomes more complex, as control dependences must be taken into account when ordering the statements of the merged program. We define a reconstitution algorithm for a language with mostly reducible control-flow, which completes the integration algorithm and allows integration of programs with more complex control-flow. This reconstitution algorithm can also be used to produce sequential code from programs written in a parallel language.

The thesis is organized as follows. Comparisons with related work are found in each chapter. Chapter 2 provides background material for profiling, tracing, and event counting. Chapter 3 considers the problems of profiling and tracing programs efficiently. Chapter 4 shows how to efficiently count events in a program's execution with support for on-line queries. The remainder of the chapters are devoted to the problems of slicing, differencing, and integrating programs with complex control-flow. Chapter 5 provides background material on program dependences needed for slicing, differencing, and integrating. Chapter 6 describes and solves the problem of slicing programs with complex control-flow. Chapter 7 presents our integration algorithm for programs with reducible control-flow. The main problem addressed in this chapter is that of reconstituting a merged program from a set of slices. Chapter 8 concludes the thesis and discusses future directions for research.

Chapter 2

BACKGROUND FOR PROFILING, TRACING, AND EVENT COUNTING

Chapters 3 and 4 present algorithms for instrumenting programs to record information about their execution-time behavior. These algorithms use the intraprocedural control-flow structure of programs in order to determine where to place instrumentation code. The programs under consideration are assumed to have been written in an imperative language with procedures, in which control-flow within a procedure is statically determinable. Interprocedural control-flow occurs mainly by procedure call and procedure return, although we will show how the algorithms can be extended to handle exceptions and interprocedural jumps. Whether or not procedures are first-class objects does not affect the instrumentation algorithms. The algorithms require only that a control-flow graph can be constructed for each procedure in the program. It is not necessary to know which procedure is called at a particular call site.

We first review some graph terminology. A directed graph $G = (V, E)$ consists of a set of vertices V and set of edges E, where an edge e is an ordered pair of vertices, denoted by $v \rightarrow w$ (note that parallel edges between vertices are allowed; the notation $v \rightarrow w$ is an abbreviation). Vertex v is the *source* of edge e, denoted by $src(e)$, and vertex w is the *target* of edge e, denoted by $tgt(e)$. Edge $v \rightarrow w$ is an *incoming* edge of vertex w and an *outgoing* edge of vertex v. If $v \rightarrow w$, then vertex v is a *predecessor* of vertex w and vertex w is a *successor* of vertex v. A *path* in a directed graph is a sequence of n vertices and $n-1$ edges of the form $(v_1, e_1, v_2, \ldots e_{n-1}, v_n)$, where for each edge e_i, either $e_i = v_i \rightarrow v_{i+1}$ or $e_i = v_{i+1} \rightarrow v_i$. A *cycle* is a path such that $v_1 = v_n$. A path or cycle is *directed* if for every edge e_i, $e_i = v_i \rightarrow v_{i+1}$. Finally, a *simple cycle* is a cycle in which $\{v_1 \cdots v_{n-1}\}$ are distinct. If a cycle is simple then the edges in the cycle are distinct, but the converse is not true.

In the next two chapters, we use the terms *path* and *cycle* to denote undirected paths and cycles. When edge direction is important we explicitly state that a path or cycle is directed.

A control-flow graph (CFG) is a rooted directed graph $G = (V, E)$ that corresponds to a procedure in a program in the following way: each vertex in V represents a basic block of instructions (a straight-line sequence of instructions) and each edge in E represents the transfer of control from one basic block to another. In addition, the CFG includes a special vertex *EXIT* that corresponds to procedure exit (return). The root vertex is the first basic block in the procedure. There is a directed path from the root to every vertex and a directed path from every vertex to *EXIT*. Finally, for the profiling algorithm, it is convenient to insert an edge $EXIT \rightarrow root$ to make the CFG strongly connected. This edge does not correspond to an actual flow of control and is not instrumented. The *EXIT* vertex has no successors other than the *root* vertex.

A vertex p is a *predicate* if there are distinct vertices a and b such that $p \rightarrow a$ and $p \rightarrow b$.

A *weighting* W of CFG G assigns a *non-negative* value (integer or real) to every edge subject to Kirchoff's flow law: for each vertex v, the sum of the weights of the incoming edges of v must equal the sum of the weights of the outgoing edges of v. The *weight* of a vertex is the sum of the weights of its incoming (outgoing) edges. The *cost* of a set of edges and/or vertices is the sum of the weights of the edges and/or vertices in the set.

An *execution* of a procedure is represented by a directed path EX through its CFG that begins at the root vertex (procedure entry) and ends at *EXIT* (procedure return). The *frequency* of a vertex v or edge e in an execution EX is the number of times that v or e appears in EX. If a vertex or edge does not appear in EX, its frequency is zero, except that for any execution, the frequency of the edge $EXIT \rightarrow root$ is defined to be the number of times that *EXIT* appears in the execution. The edge frequencies for any execution of a CFG constitute a weighting of the CFG.

Because the algorithms presented in the next two chapters depend on spanning trees, we quickly review some of the terminology of this area. Although the CFG is a directed graph, the spanning trees of the CFGs that we consider are undirected (edge direction does not matter). A *spanning tree* of a directed graph G is a subgraph $H = (V, T)$, where $T \subseteq E$, with a unique path between each pair of vertices. The edges in T are called *tree edges* while the other edges (in $E-T$) are called *chords* (of the spanning tree). The addition of a chord e to the spanning tree creates exactly one simple cycle. This cycle is called the *fundamental cycle* of e and is denoted by $C(e)$. It is important to note that fundamental cycles are always defined with respect to some spanning tree. A *maximum* spanning tree of a weighted graph is one such that the cost of the tree edges is maximal. The maximum spanning tree for a graph can be computed efficiently (linear time) by a variety of algorithms [68].

Figure 2.1 illustrates these definitions. The first graph is the CFG of the program. This graph has been given a weighting. The second graph is a maximum spanning tree of the first graph. Note that any vertex in a spanning tree can serve as a root and that the direction of the edges in the tree is unimportant. For example, vertices C and $EXIT$ are connected in the spanning tree by the path $C \rightarrow P \leftarrow EXIT$. Edge $Q \rightarrow A$ in the CFG is a chord of the spanning tree shown in the figure. The fundamental cycle associated with chord $Q \rightarrow A$ is $Q \rightarrow A \rightarrow R \rightarrow C \rightarrow P \rightarrow Q$ (which

```
program

  while P do
    if Q  then
          A
    else
          B
    fi
    if R  then  break  fi
    C
  od

end
```

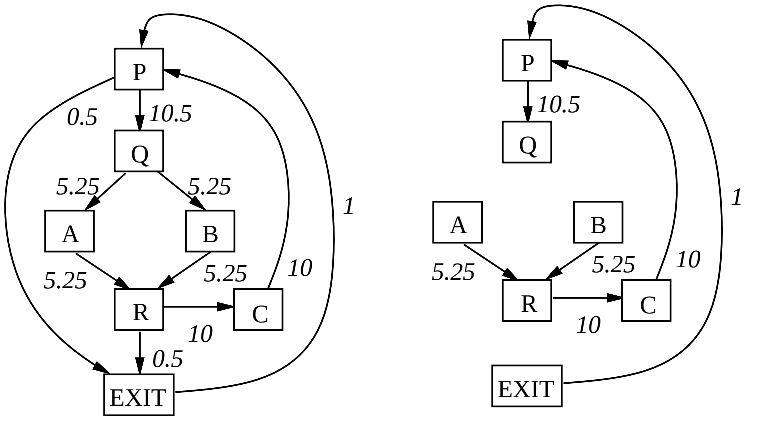


Figure 2.1. A program, its CFG with a weighting, and a maximum spanning tree. The edge $EXIT \rightarrow P$ is needed so that the flow equations for the root vertex (P) and $EXIT$ are consistent. This edge does not correspond to an actual flow of control and is not instrumented.

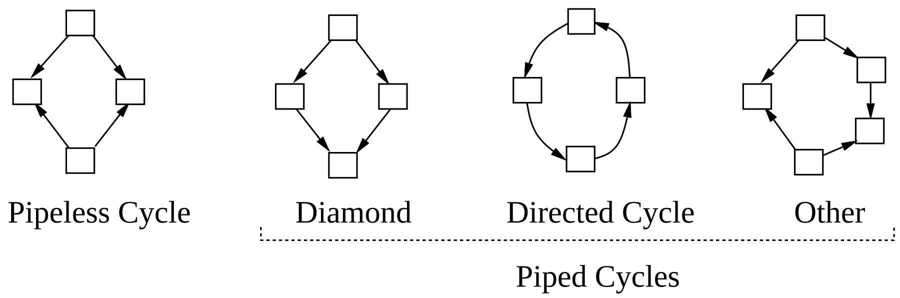


Figure 2.2. Classification of cycles.

happens to be directed, in this case).

An underlying concept in the instrumentation problems we consider is that certain cycles in a CFG must contain instrumentation code (*i.e.*, the instrumentation code must *break* certain cycles). We classify cycles based on the direction of their edges. Let u, v and w be three consecutive vertices in a cycle. There is a *fork* at v if $u \leftarrow v \rightarrow w$, a *join* if $u \rightarrow v \leftarrow w$, and a *pipe* otherwise ($u \rightarrow v \rightarrow w$ or $u \leftarrow v \leftarrow w$). A cycle is *pipeless* if it contains no pipes (*i.e*, the direction of edges strictly alternate around the cycle). A cycle is *piped* if it contains at least one pipe. Piped cycles are further classified: a *directed cycle* contains only pipes (all edges are in the same direction); a *diamond* is a cycle with more than two distinct edges that has exactly one fork and one join (there are two changes of direction in the cycle); *other* cycles are all other piped cycles. Figure 2.2 gives examples of these cycles.

Chapter 3

OPTIMALLY PROFILING AND TRACING PROGRAMS

A well-known technique for recording program behavior and measuring program performance is to insert code into a program and execute the modified program. This chapter discusses how to insert monitoring code to either profile or trace programs. Program profiling counts the number of times that each basic block or control-flow edge in a program executes. It is widely used to measure instruction set utilization, identify program bottlenecks, and estimate program execution times for code optimization [13, 19, 26, 48, 50, 53, 64]. Instruction tracing records the sequence of basic blocks traversed in a program execution. It is the basis for trace-driven architectural simulation and analysis and is also used in trace-driven debugging [11, 44, 66]. Both techniques have been implemented in a wide variety of systems.

In this chapter, we describe algorithms for placing profiling and tracing code that greatly reduce the cost of measuring programs, compared to previously implemented approaches. The algorithms reduce measurement overhead in two ways: by inserting less instrumentation code and by placing the code where it is less likely to be executed. The algorithms have been implemented in a widely-distributed profiling/tracing tool called *qpt* [45], which instruments executable files, and performs very well in practice.

The algorithms in this chapter produce an *exact* basic block profile or trace, contrasted with statistical tools such as the Unix™ *prof* command, which samples the program counter during program execution. The algorithms consist of a pre-execution phase and a post-execution phase. The first phase selects points in a program at which to insert profiling or tracing code. Instrumentation code is inserted at these points, producing an instrumented version of the program. The algorithms for inserting instrumentation for profiling and tracing are nearly identical. Both compute a spanning tree of the program's control-flow graph and place the instrumentation code on control-flow graph edges not in the spanning tree. In profiling, the instrumentation code increments a counter that records how many times an edge executes. In tracing, the instrumentation code writes a unique token (witness) to a trace file. Placement of instrumentation code can be optimized with respect to a *weighting* that orders the execution frequency of edges or vertices. Weightings can be obtained either by empirical measurement (profiling) or by estimation. After the instrumented program executes, the second phase uses the results collected during execution and the program's control-flow graph to derive a complete profile or trace.

The major contributions of this chapter are:

- We enumerate the space of profiling problems based on what is profiled and where profiling code is placed. A *vertex profile* counts the number of executions of each vertex (basic block) in a control-flow graph. An *edge profile* counts the number of times each control-flow edge executes. An edge profile determines a vertex profile, but the converse does not always hold. Knuth has published efficient algorithms for finding the minimum number of vertex counters necessary and sufficient for vertex profiling [41], denoted by *Vprof* (*Vcnt*), and the minimum number of edge counters for edge profiling [40], denoted by *Eprof* (*Ecnt*). We consider the new problem of finding a set of edge counters for vertex profiling, *Vprof* (*Ecnt*), and characterize when a set of instrumented edges is necessary and sufficient for vertex profiling.

- We relate the optimal solutions to three profiling problems, *Vprof* (*Vcnt*), *Eprof* (*Ecnt*), and *Vprof* (*Ecnt*), and compare their run-time overhead in practice. We show that for a given CFG and weighting, an optimal solution to *Vprof* (*Vcnt*) or *Eprof* (*Ecnt*) is never better than an optimal solution to *Vprof* (*Ecnt*). Unfortunately, finding an optimal solution to *Vprof* (*Ecnt*) seems to be a hard problem in general. We believe the problem is NP-complete but do not have a proof as of yet. However, we show that for a large class of structured control-flow graphs, an optimal solution to *Eprof* (*Ecnt*) is an optimal solution to *Vprof* (*Ecnt*). Furthermore, we show that *Eprof* (*Ecnt*) has lower overhead than *Vprof* (*Vcnt*) in practice.

- We show that for both profiling and tracing, placing instrumentation code on edges is better than placing it on vertices. Intuitively this is because there are more edges than vertices in the control-flow graph. Instrumenting edges provides more opportunities to place instrumentation code in areas of low execution frequency.
- We show that a simple heuristic for estimating execution frequencies (based on analysis of the control-flow graph) can accurately predict areas of low execution frequency at which to place instrumentation code.
- We show that any solution to a profiling problem is sufficient to solve the tracing problem. However, such a solution is not necessarily optimal. Ramamoorthy, Kim, and Chen have given a necessary and sufficient condition for when a set of edges solves the tracing problem for single procedure programs [57]. However, this condition does not work for multi-procedure programs. We reformulate this condition in a more intuitive manner and show how it can be extended to apply to multi-procedure programs.

Our work shows that Knuth's algorithm for *Eprof* (*Ecnt*) profiling is the algorithm of choice: It is simple and efficient, finds optimal counter placements in most cases, and yields more information than a vertex profile (by measuring edge frequency as well as vertex frequency). We show how to extend this algorithm to handle early procedure termination caused by exceptions.

We emphasize that the algorithms presented here are based solely on control-flow information. They are applicable to any control-flow graph. The graphs need not be reducible or have other properties that would preclude the analysis of some programs. The algorithms do not make use of other semantic information that could be derived from the program text (*e.g.*, via constant propagation or induction variable analysis). However, such information could be used to further improve instrumentation code.

The remainder of this chapter is organized as follows. Section 3.1 shows how to profile programs efficiently and Section 3.2 describes how to trace programs efficiently. Section 3.3 presents our heuristic weighting algorithm. Section 3.4 presents performance results. Section 3.5 reviews related work on profiling, tracing, and heuristics for minimizing instrumentation overhead and estimating execution frequency.

3.1. PROGRAM PROFILING

In order to determine how many times each basic block in a program executes, the program can be instrumented with counting code. The simplest approach places a counter at every basic block (*pixie* and other instrumentation tools use this method [67]). There are two drawbacks to such an approach: (1) too many counters are used and (2) the total number of increments during an execution is larger than necessary.

The *vertex profiling* problem, denoted by *Vprof* (*cnt*), is to determine a placement of counters *cnt* (a set of edges and/or vertices) in CFG *G* such that the frequency of each vertex in any execution of *G* can be deduced solely from the CFG *G* and the measured frequencies of edges and vertices in *cnt*. Furthermore, to reduce the cost of profiling, the set *cnt* should minimize cost for a weighting *W*.

A similar problem is the *edge profiling* problem, denoted by *Eprof* (*cnt*): determine a placement of counters *cnt* in CFG *G* such that the frequency of each edge in any execution of *G* can be deduced solely from the CFG *G* and the measured frequencies of edges and vertices in *cnt*. A solution to the edge frequency problem obviously yields a solution to the vertex frequency problem by summing the frequencies of incoming or outgoing edges of each vertex.

Given that we can place counters on vertices or edges, a counter placement can take one of three forms: a set of edges (*Ecnt*); a set of vertices (*Vcnt*); a mixture of edges and vertices (*Mcnt*). Combined with the two profiling problems, this yields six possibilities. We do not consider *Eprof* (*Vcnt*), since there are CFGs for which there are no solutions to this problem [55]. That is, it is not always possible to determine edge frequencies from vertex frequencies. Mixed placements are of interest because placing counters on vertices rather than edges eliminates the need to insert unconditional jumps.[1] On the other hand, a vertex is executed more frequently than any of its outgoing edges, implying that it might be worthwhile to instrument some outgoing edges rather than the vertex. The usefulness of mixed placements depends on the cost of an unconditional jump relative to the cost of incrementing a counter in memory. On RISC machines (for which we constructed a profiling tool) the code sequence for incrementing a counter or generating a tracing token ranges from 5 to 11 instructions (cycles). The cost of an unconditional branch is quite small in comparison (usually 1 cycle, as the delay slot of an unconditional branch can almost always be filled with a useful instruction). In this case, there is questionable benefit from mixed placements. In fact, Samples has shown that mixed placements provide little benefit over edge placements on a machine in which the increment and branch costs were comparable, and were worse in some cases [63]. Furthermore, as shown in Section 3.4.1, for all the benchmarks we examined, less than half of the instrumented edges (which is about one quarter of the total number of control-flow edges) required unconditional jumps when profiling with edge counters. For these reasons, we do not consider mixed counter placements.

We focus on the remaining three profiling problems: *Vprof* (*Vcnt*), *Eprof* (*Ecnt*), and *Vprof* (*Ecnt*). This section presents four results:

(1) A comparison of the optimal solutions to *Vprof* (*Vcnt*), *Eprof* (*Ecnt*), and *Vprof* (*Ecnt*). Figure 3.1(a) summarizes the relationship between these three problems for general CFGs. $X \leq Y$ means that for any given CFG and weighting, an optimal solution to problem X has cost less than or equal to the cost of an optimal solution to problem Y. In general, for any weighted CFG, an optimal solution to *Vprof* (*Ecnt*) is always at least as cheap as *Eprof* (*Ecnt*) or *Vprof* (*Vcnt*).

---

[1]Placing instrumentation code along edges of the CFG essentially creates new basic blocks, which may require the insertion of unconditional jumps (assuming that the linearization of the original basic blocks is the same in the instrumented program as in the original program). On the other hand, placing instrumentation code in vertices simply expands the extent of the original basic blocks, and does not require insertion of jumps. It is possible to rearrange the placement of basic blocks to minimize the number of unconditional jumps needed, as discussed by Ramanath and Solomon [58]. However, our algorithms do not perform such an optimization, as they respect the original linearization.

(2) A characterization of when a set of edges *Ecnt* is necessary and sufficient for *Eprof* (*Ecnt*), and an algorithm to solve *Eprof* (*Ecnt*) optimally. We also describe the problem introduced by early procedure termination and a simple solution.

(3) A characterization of when a set of edges *Ecnt* is necessary and sufficient for *Vprof* (*Ecnt*). However, it appears difficult to efficiently find a minimal size or cost set of such edges. We show that an optimal solution to *Eprof* (*Ecnt*) is also an optimal solution to *Vprof* (*Ecnt*) for a large class of structured CFGs and present a heuristic for solving *Vprof* (*Ecnt*) using the *Eprof* (*Ecnt*) algorithm as a subcomponent.

(4) A discussion of the time complexity of the profiling and tracing problems, based on their characterization as cycle breaking problems.

3.1.1. Comparing the Three Profiling Problems

This section examines the relationships between the optimal solutions to *Vprof* (*Vcnt*), *Eprof* (*Ecnt*), and *Vprof* (*Ecnt*) for general CFGs, as summarized in Figure 3.1(a).

The three CFGs in Figure 3.2 illustrate optimal solutions to *Vprof* (*Vcnt*), *Eprof* (*Ecnt*), and *Vprof* (*Ecnt*) (for the weighting given in the first CFG). The black dots represent counters. The costs of the three counter placements are 124, 62 and 59, respectively. In each case, every

(a) *Eprof(Ecnt)*   *Vprof(Vcnt)*

Vprof(Ecnt)

(b) $Eprof(Ecnt) = Vprof(Ecnt) \leqslant Vprof(Vcnt)$

Figure 3.1. (a) The relationship between the costs of the optimal solutions of the three frequency problems for general CFGs. (b) The relationship when the CFGs are constructed from **while** loops, **if-then-else** conditionals, and **begin-end** blocks.

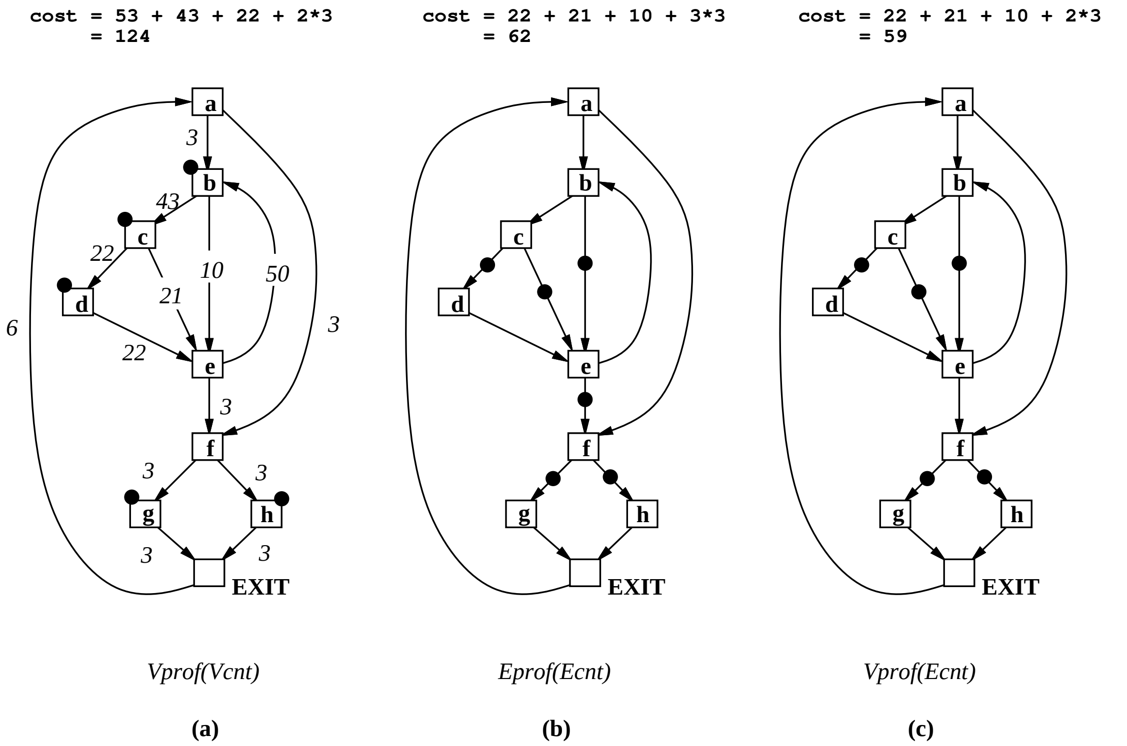


Figure 3.2. Optimal solutions for (a) vertex profiling with vertex counters, (b) edge profiling with edge counters and (c) vertex profiling with edge counters.

counter is necessary to uniquely determine a profile and no lower cost placements will suffice. For example, if the counter on vertex *b* in case (a) were eliminated, it would be impossible to determine how many times *b* or *e* executed. In case (a), the counts for vertices *a*, *e*, *f*, and *EXIT* are not directly measured, but can be deduced from the measured vertices as follows: $e = b$; $a = f = EXIT = g + h$. In case (b), the count for each unmeasured edge is uniquely determined by the counts for the measured edges by Kirchoff's flow law (*e.g.*, $a \to f = f \to g + f \to h - e \to f$). In case (c), the count for each unmeasured edge except those in the set $\{ a \to b, e \to b, e \to f, a \to f \}$ is uniquely determined by the measured edges. This yields enough information to deduce the count for each vertex.

For any CFG and weighting, an optimal solution to *Vprof* (*Vcnt*) never has lower cost than an optimal solution to *Vprof* (*Ecnt*) (for every vertex v in *Vcnt*, v's counter can be "pushed" off v onto each outgoing edge of v, resulting in counter placement *Ecnt*, which clearly solves the vertex profiling problem with cost equal to *Vcnt*). Figure 3.2 shows an example where *Vprof* (*Ecnt*) has lower cost than *Vprof* (*Vcnt*). The counter placement in case (c) solves *Vprof* (*Ecnt*) and has lower cost than the counter placement in case (a) that solves *Vprof* (*Vcnt*).

Since any solution to *Eprof* (*Ecnt*) must also solve *Vprof* (*Ecnt*), an optimal solution to *Eprof* (*Ecnt*) can never have lower cost than an optimal solution to *Vprof* (*Ecnt*), for a given CFG and weighting. The counter placement in case (c) solves *Vprof* (*Ecnt*) and has lower cost than the counter placement in case (b) that solves *Eprof* (*Ecnt*). In comparing *Eprof* (*Ecnt*) and *Vprof* (*Vcnt*), there are examples in which one has lower cost than the other and vice versa. Cases (b) and (a) of Figure 3.2 show an example where *Eprof* (*Ecnt*) has lower cost than *Vprof* (*Vcnt*). Figure 3.2(c) can be easily modified to show an example where *Vprof* (*Vcnt*) has lower cost than *Eprof* (*Ecnt*). Consider each black dot as a vertex in its own right and split the dotted edge into two edges. The dots constitute the set *Vcnt* and solve *Vprof* (*Vcnt*) with cost 59. The optimal solution to *Eprof* (*Ecnt*) for this graph still has cost 62.

3.1.2. Edge Profiling with Edge Counters

Eprof (*Ecnt*) can be solved by placing a counter on the outgoing edges of each predicate vertex. However, this placement uses more counters than necessary. Knuth describes how it follows from Kirchoff's law that an edge-counter placement *Ecnt* solves *Eprof* (*Ecnt*) for CFG $G = (V,E)$ *iff* $(E - Ecnt)$ contains no (undirected) cycle [40]. Since a spanning tree of a CFG represents a maximum subset of edges without a cycle, it follows that *Ecnt* is a minimum size solution to *Eprof* (*Ecnt*) *iff* $(E - Ecnt)$ is a spanning tree of G. Thus, the minimum number of counters necessary to solve *Eprof* (*Ecnt*) is $|E| - (|V| - 1)$.

To see how such a placement solves the edge frequency problem, consider a CFG G and a set *Ecnt* such that $E - Ecnt$ is a spanning tree of G. Let each edge e in *Ecnt* have an associated counter that is initially set to 0 and is incremented once each time e executes. If vertex v is a leaf in the spanning tree (*i.e.*, only one tree edge is incident to v), then all remaining edges incident to v are in *Ecnt*. Since the edge frequencies for an execution satisfy Kirchoff's law, the unmeasured edge's frequency is uniquely determined by the flow equation for v and the known frequencies of the other incoming and outgoing edges of v. The remaining edges with unknown frequency still form a tree, so this process can be repeated until the frequencies of all edges in $E - Ecnt$ are uniquely determined. If $E - Ecnt$ contains no cycles but is not a spanning tree, then $E - Ecnt$ is a forest of trees. The above approach can be applied to each tree separately to determine the frequencies for the edges in $E - Ecnt$.

Any of the well-known maximum spanning tree algorithms described by Tarjan [68] will efficiently find a maximum spanning tree of CFG G with respect to weighting W. The edges that are not in the spanning tree solve $Eprof\,(Ecnt)$ and minimize the cost of *Ecnt*. As a result, counters are placed in areas of lower execution frequency in the CFG. To ensure that a counter is never placed on $EXIT \rightarrow root$, the maximum spanning tree algorithm can be seeded with the edge $EXIT \rightarrow root$. In fact, for any CFG and weighting, there is always a maximum spanning tree that includes the edge $EXIT \rightarrow root$. The derived count for the edge $EXIT \rightarrow root$ represents the number of times the procedure associated with CFG G executed.

Figure 3.3(a) illustrates how the frequencies of edges in $E - Ecnt$ can be derived from the frequencies of edges in *Ecnt*. Black dots identify edges in *Ecnt*. The other edges are in $E - Ecnt$ and form a spanning tree of the CFG. The edge frequencies are those for the execution shown. However, we emphasize that the only edges for which frequencies will be recorded are the edges with black dots. Let vertex P be the root of the spanning tree. Vertex Q is a leaf in the spanning tree and has flow equation ($P \rightarrow Q = Q \rightarrow A + Q \rightarrow B$). Since the frequencies for $P \rightarrow Q$ and $Q \rightarrow A$ are known, we can substitute them into this equation and derive the frequency for $Q \rightarrow B$. Once the frequency for $Q \rightarrow B$ is known, the frequency for $B \rightarrow R$ can be derived from the flow equation for B, and so on. For the weighting W given in Figure 2.1, the solution in Figure 3.3(a) has cost 16.75. However, Figure 3.3(b) shows a solution based on the *maximum* spanning tree with resultant cost of 11.5.

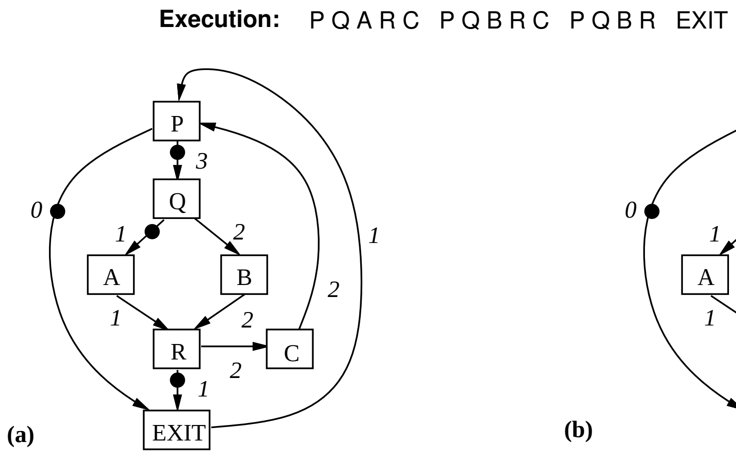


Figure 3.3. Solving *Eprof* (*Ecnt*) using the spanning tree. For the weighting given in Figure 2.1, the counter placement in case (a) is not optimal (minimal) but the counter placement in case (b) is optimal.

The propagation algorithm in Figure 3.4 performs a post-order traversal of the spanning tree $E - Ecnt$ to propagate the frequencies of edges in *Ecnt* to the unprofiled edges in the spanning tree. The procedure DFS calculates the frequency of a spanning tree edge. Since the calculation is carried out post-order, once the last line in DFS(G, $Ecnt$, v, e) is reached, the counts of all edges incident to vertex v except e have been calculated. The flow equation for v states that the sum of v's incoming edges is equal to the sum of v's outgoing edges. One of these sums includes the count from edge e, which has been initially set to 0. The count for e is found by subtracting the minimum of the two sums from the maximum.

Although profiling has been described in terms of a single CFG, the algorithm requires few changes to deal with multi-procedure programs. The pre-execution spanning tree algorithm and post-execution propagation of edge frequencies can be applied to each procedure's CFG separately. This simple extension for multi-procedure profiling will determine the correct frequencies whenever interprocedural control-flow occurs only via procedure call and return and

```
global
    G:  control-flow graph
    E:  edges of G
    cnt:  array[edge] of integer;      /* for each edge e in Ecnt, cnt[e] = frequency of e in execution */

procedure propagate_counts(Ecnt: set of edges)
begin
    for each e ∈ E − Ecnt do cnt[e] := 0 od
    DFS(Ecnt, root-vertex(G), NULL)
end

procedure DFS(Ecnt: set of edges; v: vertex; e: edge)
let   IN(v) = { e′ | e′∈ E and v = tgt(e′) }   and   OUT(v) = { e′ | e′∈ E and v = src(e′) }   in
    in_sum := 0;
    for each e′ ∈ IN(v) do
        if (e′ ≠ e) and e′∈ E − Ecnt then DFS(Ecnt, src(e′), e′) fi
        in_sum := in_sum + cnt[e′]
    od
    out_sum := 0;
    for each e′ ∈ OUT(v) do
        if (e′ ≠ e) and e′∈ E − Ecnt then DFS(Ecnt, tgt(e′), e′) fi
        out_sum := out_sum + cnt[e′]
    od
    if e ≠ NULL then cnt[e] := max(in_sum, out_sum) − min(in_sum, out_sum) fi
ni
```

Figure 3.4. Edge propagation algorithm determines the frequencies of edges in the spanning tree *E-Ecnt* given the frequencies of edges in *Ecnt*. The algorithm uses a post-order traversal of the spanning tree.

each call eventually has a corresponding return.[2] Statically-determinable interprocedural jumps (other than procedure call and return) can be handled by adding edges corresponding to the interprocedural jumps and instrumenting these edges. Determining whether or not such an interprocedural edge needs to be instrumented would require interprocedural analysis that we did not perform.

A problem arises with dynamically computed interprocedural jumps such as `setjmp/longjmp` in the *C* language [38], or early program termination, as may be caused by a

[2]For the purposes of determining the frequencies of intraprocedural control-flow edges, it does not matter whether procedures and functions are first class objects. For programs with a fixed call graph structure, the intraprocedural frequency information is sufficient to determine the frequency of edges in the call graph. For programs with procedure or function parameters, a tool must record the callee at call sites at which the callee is determined at run-time.

system call or an error condition. In these cases, one or more procedures terminate before reaching the *EXIT* vertex, breaking Kirchoff's law. For example, suppose that the CFG in Figure 3.5(a) executes the path shown at the top of the figure. Furthermore, suppose that the execution terminates early at vertex *A* because of a divide by zero error. As a result, control enters vertex *A* once via the edge $Q \rightarrow A$ once but never exits via $A \rightarrow R$. However, because the propagation algorithm (see Figure 3.4) assumes that Kirchoff's law holds at each vertex, edge $A \rightarrow R$ will receive a count of 1, as shown in Figure 3.5(a). In this example, the count is off by one. However, in general, if multiple procedures on the activation stack are exited early and early exiting is a common occurrence, the counts may diverge greatly.

In this case, information available on the activation stack is sufficient to correct the count error. Conceptually, for each procedure *X* on the activation stack that exits early an edge $v \rightarrow EXIT$ with a count of 1 is added to procedure *X*'s CFG, where *v* is the vertex from which procedure *X* called the next procedure. This edge models early termination of procedure *X* at vertex *v*. In practice,

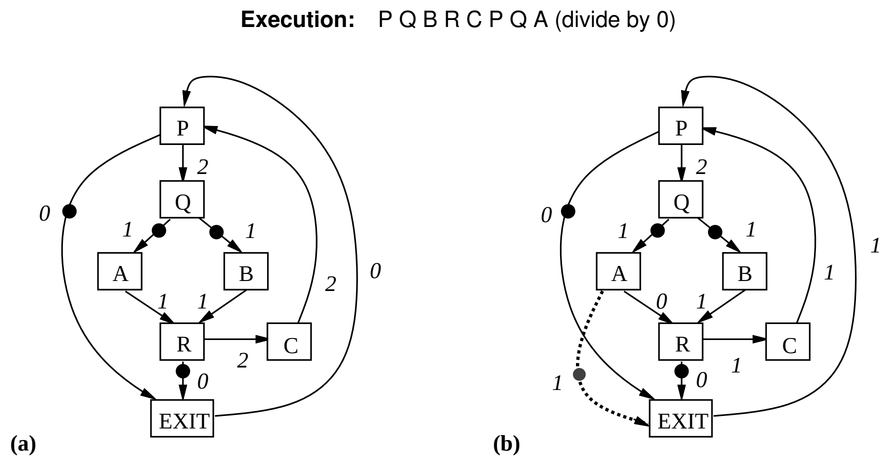


Figure 3.5. (a) Early termination at vertex *A* yields incorrect counts, (b) which are corrected by the addition of edge $A \rightarrow EXIT$.

the edge $v \rightarrow EXIT$ is represented by an "exit" counter that is associated with the vertex v. This counter is incremented once for each time procedure X exits early when at vertex v. For early termination caused by a conditional exception (such as divide by zero) the increment code must be placed in the exception handler rather than at vertex v, since the code should only be invoked only when v raises the exception. For early termination caused by `longjump`, the increment code must also be in the handler since `longjump` may pop many activation frames off the stack, each of which requires incrementing the associated exit counter.

Figure 3.5(b) illustrates how the early exit problem is solved. Because the procedure terminates early at vertex A, the edge $A \rightarrow EXIT$ is added to the CFG and given a count of 1. This additional edge correctly siphons off the incoming flow to vertex A so that the propagation algorithm yields correct counts. As shown in case(b), edge $A \rightarrow R$ correctly receives a count of 0.

3.1.3. Vertex Profiling with Edge Counters

This section addresses the problem of vertex profiling with edge counters. Section 3.1.3.1 characterizes when a set of edges *Ecnt* solves *Vprof* (*Ecnt*) and gives an algorithm for propagating edge frequencies through the CFG in order to determine vertex frequencies. As discussed later in Section 3.1.4, it appears difficult to solve *Vprof* (*Ecnt*) efficiently while minimizing the size or cost of *Ecnt*. However, as discussed in Section 3.1.3.2, there are certain classes of CFGs for which an optimal solution to *Eprof* (*Ecnt*) is also an optimal solution to *Vprof* (*Ecnt*). For this class of CFGs, the counter placements induced by the maximum spanning tree are optimal. Finally, Section 3.1.3.3 presents a heuristic for finding an *Ecnt* placement to solve *Vprof* (*Ecnt*) that improves on the spanning tree approach in certain situations.

3.1.3.1. Characterization and algorithm

Edge profiling with edge counters requires that every (undirected) cycle in the CFG contain a counter. Since an edge profile determines a vertex profile, vertex profiling requires no more edge counters than does edge profiling. However, as illustrated by the example in Figure 3.2(c), there are cases in which fewer edge counters are needed for vertex profiling than for edge profiling. In this example, there is a cycle of counter-free edges, yet there is enough information recorded to determine the frequency of every vertex. This section formalizes this observation. That is, certain types of counter-free cycles are allowed when using edge counters for vertex profiling, as captured by the following theorem:

THEOREM. A set of edges *Ecnt* solves $Vprof\,(Ecnt)$ for CFG $G = (V, E)$ iff each simple cycle in $E - Ecnt$ is pipeless (*i.e.*, edges in any simple cycle in $E - Ecnt$ alternate directions).

Pipeless cycles are allowed in $E - Ecnt$ as well as non-simple piped cycles, as long as the simple cycles that compose it are pipeless. In Figure 3.2(c), the counter-free cycle comprised by the set of edges { $a \rightarrow b$, $e \rightarrow b$, $e \rightarrow f$, $a \rightarrow f$ } is pipeless. In Figure 3.7(a), the counter-free edges contain a piped cycle; however, the cycle is not simple. Both simple counter-free cycles in this example are pipeless.

Let *freq* be the function mapping edges in a CFG to their frequency in an execution. We give an algorithm that (given the frequencies of edges in *Ecnt* in the execution and the assumption that $E - Ecnt$ contains no simple piped cycle) will find a function $freq\,'$ from edges to frequencies that is *vertex-frequency equivalent* to *freq*. That is, for any vertex *v* the sum of the frequencies of *v*'s incoming (outgoing) edges under $freq'$ is the same as under *freq*. We first explain the algorithm and show how it operates on an example. We then prove the correctness of the algorithm, showing that if $E - Ecnt$ contains no simple piped cycle then *Ecnt* solves $Vprof\,(Ecnt)$. Finally, we show that if $E - Ecnt$ contains a simple piped cycle then it is not possible for *Ecnt* to solve $Vprof\,(Ecnt)$.

Figure 3.6 presents the propagation algorithm. The frequencies for edges in *Ecnt* have been determined by an execution *EX*. The algorithm operates as follows: while there is a (simple) cycle *C* in the set of edges $E - (Ecnt \cup Break)$, an edge *e* from cycle *C* is added to the set *Break* and the frequency of edge *e* is initialized to zero. Once $E - (Ecnt \cup Break)$ is acyclic, it follows that the frequencies of edges in $Ecnt \cup Break$ uniquely determine the frequencies of the other edges (by the spanning tree propagation algorithm, as given in Figure 3.4). As we will show, the vertex frequencies determined by these edge frequencies are the true vertex frequencies in the

```
/* Assumption: E − Ecnt contains no simple piped cycle */
/* for each edge e in Ecnt, cnt[e] = frequency of e in execution */

Break := ∅
while there is a simple cycle C in E −(Ecnt ∪ Break) do
   let e be an edge in C in
      Break := Break ∪ { e };
      cnt[e] := 0;
   ni
od

propagate_counts(Ecnt ∪ Break)    /* from Figure 3.4 */
```

Figure 3.6. Algorithm for propagating edge counts to determine vertex counts.

execution *EX*.

Figure 3.7 presents an example of how this algorithm works. The CFG in Figure 3.7(a) contains two simple cycles in $E - Ecnt$. As usual, edges in *Ecnt* are marked with black dots. Each of the counter-free simple cycles is clearly pipeless. These two simple cycles combine into a non-simple cycle containing a pipe, which is allowed under the structural characterization of *Vprof* (*Ecnt*). The edges in the CFG are numbered with their frequencies from some execution. The frequencies of the checked edges can be derived easily from the frequencies of the edges in *Ecnt*. From these frequencies, the count of every vertex except the grey vertex can clearly be determined. How do we derive counts for the edges in the two simple pipeless cycles in order to determine the frequency of the grey vertex? Suppose the algorithm chooses to break the two simple cycles in $E - Ecnt$ by putting the dashed edges (see Figure 3.7(b)) into the set *Break*, giving both frequency 0, as shown in case (b). Spanning tree propagation of edge frequencies in the set $Ecnt \cup Break$ to edges in $E - (Ecnt \cup Break)$ will assign unique frequencies to the other edges in the simple pipeless cycles, as shown in case (b). The sum of the frequencies of the incoming (outgoing) edges to the grey vertex is 2, which is the correct frequency (even though the frequencies of edges in the pipeless cycle are not the same as in the execution).

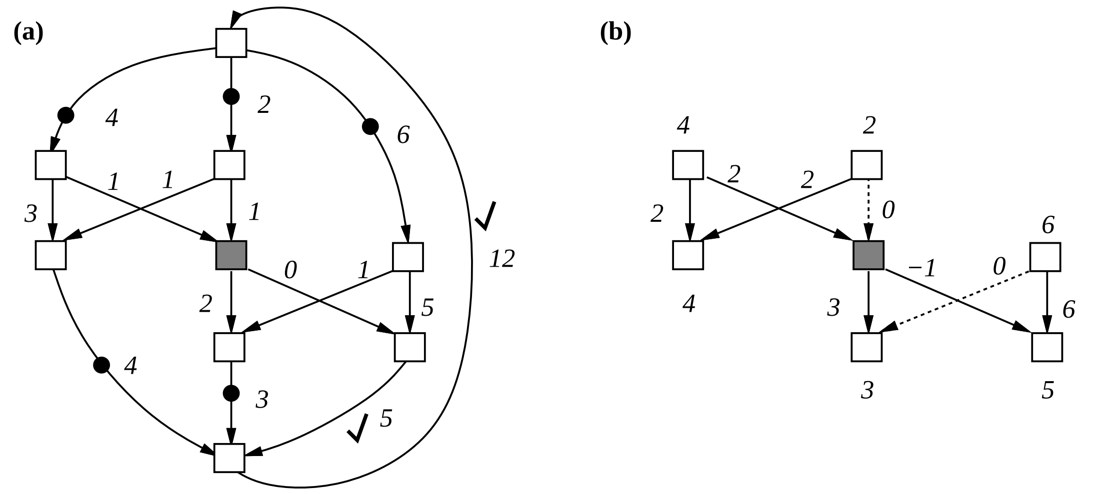


Figure 3.7. (a) An example CFG in which *E-Ecnt* contains two simple pipeless cycles. (b) If the dashed edges are assigned frequency 0, spanning tree propagation will assign the remaining edges in the simple pipeless cycles the frequencies shown. This yields a count of two for the grey vertex, which is its correct frequency.

We now prove the correctness of the algorithm. Let *freq* be the function mapping edges in a CFG to their frequency in an execution, and let *freq′* be the function from edges to frequencies created by the algorithm of Figure 3.6. We show that *freq′* is vertex-frequency equivalent to *freq* by induction on the size of *Break* (as determined by the algorithm).

Base case: $|Break| = 0$. In this case, $E - Ecnt$ contains no cycles. Therefore, *Ecnt* solves *Eprof* (*Ecnt*), so $freq' = freq$. It follows directly that *freq′* is vertex-frequency equivalent to *freq*.

Induction Hypothesis: If $|Break| \le n$ then *freq′* is vertex-frequency equivalent to *freq*.

Induction Step: Suppose that $|Break| = n+1$. Consider taking an edge *e* from *Break* and putting it in *Ecnt*, resulting in sets $Break_n$ and $Ecnt_n$. By the Induction Hypothesis, the function $freq_n$ (defined by $Break_n$ and $Ecnt_n$) is vertex-frequency equivalent to *freq*. We show that function *freq′* is vertex-frequency equivalent to $freq_n$, completing the proof. Let $T = E-(Ecnt \cup Break)$. The addition of edge *e* to *T* creates a simple pipeless cycle *C* in *T*. We define a function *g*, based on function $freq_n$, edge e, and cycle C, as shown below. We show that function g has three properties:

(1) Function g is vertex-frequency equivalent to $freq_n$;

(2) Function g satisfies Kirchoff's flow law at every vertex;

(3) For each edge $f \in Ecnt \cup Break$, $g(f) = freq'(f)$.

Points (2) and (3) imply that g and $freq'$ are identical functions (because the values of edges in $Ecnt \cup Break$ uniquely determine the values of all other edges by Kirchoff's flow law). Therefore, point (1) implies that $freq'$ is vertex-frequency equivalent to $freq_n$. The function g is defined as follows:

$$g(f) = \begin{cases} freq_n(f) & \text{if edge } f \text{ is not in cycle } C \\ freq_n(f) - freq_n(e) & \text{if edge } f \text{ is in cycle } C\text{, in the same direction as edge } e \\ freq_n(f) + freq_n(e) & \text{otherwise} \end{cases}$$

We first show that Kirchoff's flow law holds at every vertex under g and that g is vertex-frequency equivalent to $freq_n$. This is obvious for vertices that are not in C (since the frequency of any edge incident to such a vertex is the same under g and $freq_n$). Because every vertex v in C either appears in a fork or join in the cycle, one of the edges incident to v will have $freq_n(e)$ subtracted from its frequency and the other will have $freq_n(e)$ added to its frequency, thus preserving the flow law and vertex frequency at v.

We now prove point (3). It is clear that $g(e) = 0 = freq'(e)$. We must show that for each edge $f \in Ecnt \cup Break_n$, $g(f) = freq'(f)$. By definition, for each edge $f \notin C$, $g(f) = freq_n(f)$. Cycle C contains no edges from $Ecnt \cup Break_n$. Since $freq'(f) = freq_n(f)$ for all edges in $Ecnt \cup Break_n$, it follows that for each such edge f, $g(f) = freq'(f)$.

□

If $E - Ecnt$ contains a simple piped cycle, then there are two executions of G with different frequencies for some vertex but for which the frequencies of edges in $Ecnt$ are the same. This is clear if $E - Ecnt$ contains a directed cycle, or two edge-disjoint directed paths between a pair of vertices (*i.e.*, a diamond). Figure 3.8 gives an example of a CFG in which $E - Ecnt$ contains a piped cycle (the pipe is at vertex B) that is neither a directed cycle nor a diamond and shows two different execution paths. Both execution paths traverse each instrumented edge (x,y,z) exactly once. However, EX_1 contains vertex B while EX_2 does not.

Another way to look at this is that the edge frequencies in a cycle in $E - Ecnt$ are unconstrained. Let *freq* be a function mapping edges to values that satisfies Kirchoff's flow law at every vertex. Applying the function transformation defined earlier to *freq* based on a piped cycle in $E - Ecnt$ results in function $freq'$ such that Kirchoff's flow law holds at every vertex. While the frequency of each vertex in a fork or join in the cycle remains the same (as shown above), the frequency of the vertex in the pipe will have changed.

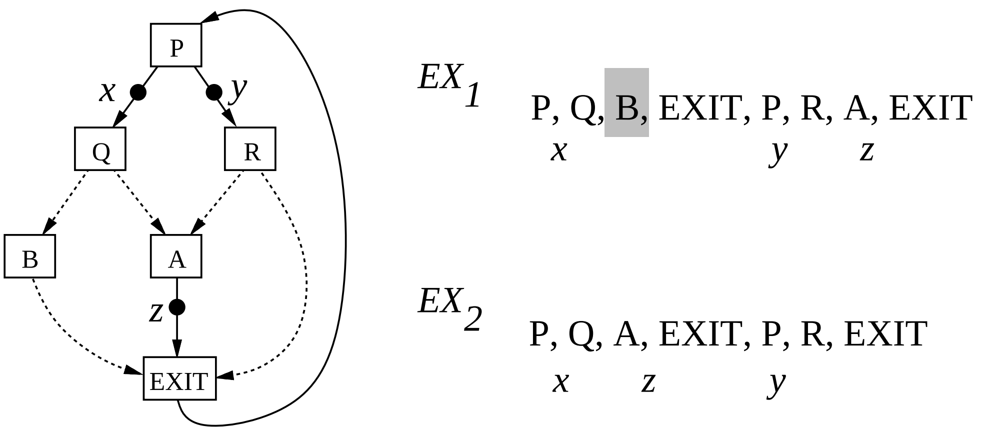


Figure 3.8. An example of instrumentation that is not sufficient for vertex profiling. The dashed edges in the CFG constitute a simple cycle of uninstrumented edges with a pipe (at vertex B). Executions EX_1 and EX_2 traverse each instrumented edge the same number of times but EX_1 contains B and EX_2 does not.

3.1.3.2. Cases for which *Eprof* (*Ecnt*) = *Vprof* (*Ecnt*)

This section examines a class of CFGs for which *Vprof* (*Ecnt*) can be solved optimally, namely those for which an optimal solution to *Eprof* (*Ecnt*) is also an optimal solution to *Vprof* (*Ecnt*). Let G^* represent all CFGs in which every cycle contains a pipe. For any CFG G in G^* with weighting W, the following statements are equivalent:

(1) *Ecnt* is a minimal cost set of edges such that $E - Ecnt$ contains no simple piped cycle;

(2) $E - Ecnt$ is a maximum spanning tree of G.

It follows directly from these two observations that for any CFG in G^*, an optimal solution to *Eprof* (*Ecnt*) is also an optimal solution to *Vprof* (*Ecnt*). The class of graphs G^* contains CFGs with multiple exit loops (such as in Figure 2.1), CFGs that can only be generated using **gotos**, and even some irreducible graphs. The class G^* contains those structured CFGS generated by **while** loops, **if-then-else** conditionals, and **begin-end** blocks (because every simple cycle in these CFGs is either a directed cycle or a diamond). However, in general, CFGs of programs with **repeat-until** loops or **break**s are not always members of G^*. The CFG in Figure 3.2 is an example of such a graph.

3.1.3.3. Heuristic for *Vprof* (*Ecnt*)

Because we believe *Vprof* (*Ecnt*) is a hard problem to solve optimally, we developed a heuristic for *Vprof* (*Ecnt*). Our heuristic first computes a maximum spanning tree *ST* and then checks if any counters can be removed (from the set of chord edges associated with *ST*) without creating simple piped cycles in the set of counter-free edges. An algorithm for the heuristic is given in Figure 3.9.

The heuristic examines each fundamental cycle $C(e)$ associated with counter edge (chord) e in turn. To prevent two pipeless cycles from combining into a simple piped cycle, we mark all vertices in the cycle $C(e)$ when a counter is removed from e; we remove a counter from an edge e only if $C(e)$ is pipeless and contains no marked vertices. The heuristic is described in detail in Figure 3.9. Upon termination, the set “Remove” contains all edges whose counters can be removed safely. By considering edges in decreasing order of weight, the algorithm tries to

```
Remove := ∅
unmark all vertices in G
find a maximum spanning tree ST of G
for each edge e ∉ ST (in decreasing order in weight) do
    if (C(e) is pipeless) and (no vertex in C(e) is marked) then
            mark each vertex in C(e)
            Remove := Remove ∪ { e }
    fi
od
```

Figure 3.9. A heuristic for *Vprof* (*Ecnt*).

remove counters with high cost first.

Consider the application of the heuristic to the CFG in Figure 3.2. Case (b) shows the counter placement resulting from the maximum spanning tree algorithm. Removing the counter on edge $e \rightarrow f$ creates a pipeless cycle in the set of counter-free edges. Removing the counter from any other edge creates a piped cycle in the set of counter-free edges. In this example, the heuristic produces the optimal counter placement in case (c). However, there are examples for which this heuristic will not find an optimal solution to *Vprof* (*Ecnt*).

3.1.4. Cycle Breaking Problems

The problems of profiling and tracing programs with edge instrumentation can be described as cycle breaking problems, where certain types of cycles in the CFG must contain instrumentation code in order to solve a profiling or tracing problem. Figure 3.10 summarizes the classification of cycles presented in Chapter 2, the problems they correspond to, and the known time complexity for (optimally) breaking each class of cycle. Solving *Eprof* (*Ecnt*) corresponds to breaking all undirected cycles. Solving *Vprof* (*Ecnt*) corresponds to breaking all simple piped cycles, as we have shown in Section 3.1.3. Finally, as discussed in Section 3.2, solving the tracing problem corresponds to breaking all directed cycles and diamonds.

Of course, we are interested in a minimum cost set of edges that breaks a certain class of cycles. Finding a minimum size set of edges that breaks all directed cycles is an NP-complete

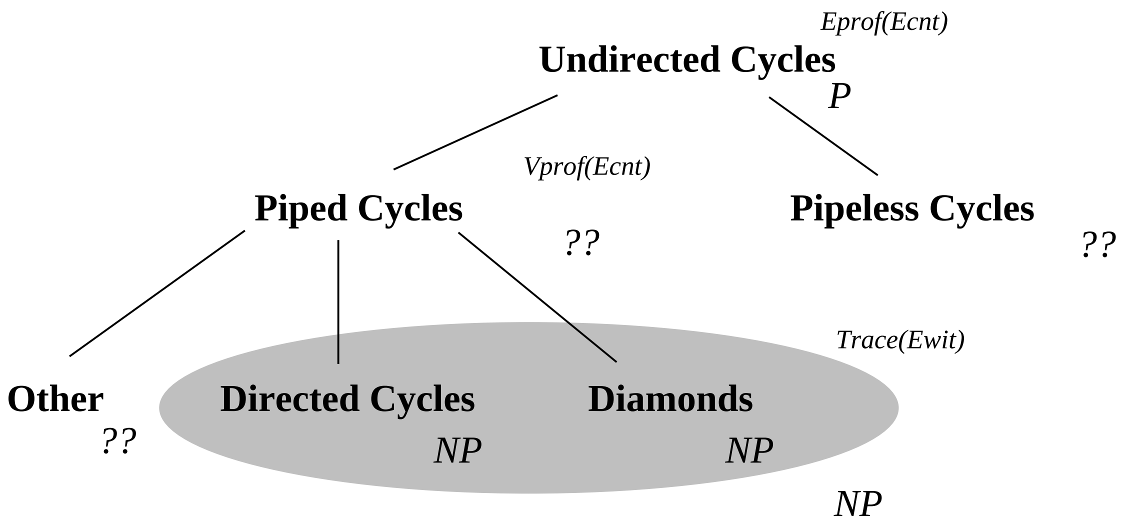


Figure 3.10. Hierarchy of cycles, the profiling or tracing problems they correspond to, and time complexity for breaking all cycles of a given type (*P* = polynomial; *NP* = NP-complete; ?? = unknown).

problem (Feedback Arc Set [23]). Maheshwari showed that finding a minimum size set of edges that breaks diamonds is also NP-complete (Uniconnected Subgraph [23, 47]). Minimizing with respect to a weighting (that satisfies Kirchoff's flow law) does not make either of these problems easier. Furthermore, it is easy to show that optimally breaking both directed cycles and diamonds is no easier than either problem in isolation. Solving the tracing problem so that the cost of the instrumented edges is minimized is an NP-complete problem, as shown in an unpublished result by S. Pottle [54]. The reduction is similar to that used by Maheshwari but is complicated by the requirement that a weighting satisfies Kirchoff's flow law.

We believe that optimally solving *Vprof* (*Ecnt*) (minimizing the size or cost of *Ecnt*) is an NP-complete problem, but do not have a proof as of yet. We have shown that a related problem, finding a minimum size set of edges that breaks all pipes, is NP-complete. Breaking all pipes guarantees that all piped cycles will be broken, but not necessarily optimally (as it is possible to break all piped cycles and still have a pipe).

3.2. PROGRAM TRACING

Just as a program can be instrumented to record basic block execution frequency, it also can be instrumented to record the sequence of executed basic blocks. The *tracing problem* is to record enough information about a program's execution to reproduce the entire execution. A straightforward way to solve this problem is to instrument each basic block so whenever it executes, it writes a unique token (called a *witness*) to a trace file. In this case, the trace file need only be read to regenerate the execution. A more efficient method is to write a witness only at basic blocks that are targets of predicates [44]. The following code regenerates the execution from a predicate trace file and the program's CFG *G*:

```
pc := root-vertex(G);
output(pc);
do
    if not IsPredicate(pc) then pc := successor(G, pc)
    else pc := read(trace) fi
    output(pc);
until ( pc = EXIT )
```

Assuming a standard representation for witnesses (*i.e.,* a byte, half-word, or word per witness), the tracing problem can be solved with significantly less time and storage overhead than the above solution by writing witnesses when edges are traversed (not when vertices are executed) and carefully choosing the witnessed edges. Section 3.2.1 formalizes the tracing problem for single-procedure programs. Section 3.2.2 considers complications introduced by multi-procedure programs.

3.2.1. Single-Procedure Tracing

In this section, assume basic blocks do not contain calls and that the extra edge $EXIT \rightarrow root$ is not included in the CFG. The set of instrumented edges in the CFG is denoted by *Ewit*. For tracing, whenever an edge in *Ewit* is traversed, a "witness" to that edge's execution is written to a trace file. We assume that no two edges in *Ewit* generate the same witness, although this is stronger than necessary as it may be possible to reuse witnesses in some cases. The statement of the tracing problem relies on the following definitions:

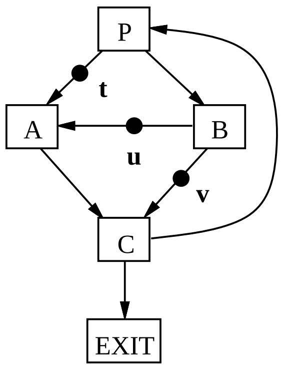
DEFINITION. A path in CFG G is *witness-free* with respect to a set of edges *Ewit* iff no edge in the path is in *Ewit*.

DEFINITION. Given a CFG G, a set of edges *Ewit*, and edge $p \rightarrow q$ where p is a predicate, the *witness set* (to vertex q) for predicate p is:

$$\begin{aligned} &witness(G,\ Ewit,\ p,\ q)\ = \\ &\quad \{\ w \mid\ p \rightarrow q\ \in\ Ewit \text{ (and writes witness } w\text{)}\ \} \\ &\cup\ \{\ w \mid\ x \rightarrow y\ \in\ Ewit \text{ (and writes witness } w\text{) and } \exists \text{ witness-free path } p \rightarrow q \rightarrow \cdots \rightarrow x\ \} \\ &\cup\ \{\ EOF \mid\ \exists \text{ witness-free path } p \rightarrow q \rightarrow \cdots \rightarrow EXIT\ \} \end{aligned}$$

Figure 3.11 illustrates these definitions. We use $witness(p,\ q)$ as an abbreviation for $witness(G,\ Ewit,\ p,\ q)$.

Let us examine how the execution in Figure 3.11 can be regenerated from its trace. Re-execution starts at predicate P, the root vertex. To determine the successor of P, we read witness



Execution: P A C P B A C P B C EXIT

Trace: **t** **u** **v** **EOF**

witness(P, A) = **{ t }**
witness(P, B) = **{ u, v }**

witness(B,A) = **{ u }**
witness(B, C) = **{ v }**

witness(C, P) = **{ t, u, v }**
witness(C, EXIT) = **{ EOF }**

Figure 3.11. Example of a traced function. Vertices P, B, and C are predicates. The witnesses are shown by labeled dots on edges. For the execution shown, the trace generated is **(t, u, v, EOF)**. The witness **EOF** is always the last witness in a trace. The execution can be reconstructed from the trace using the witness sets to guide which branches to take.

t from the trace, which is a member of $witness(P,A)$ but not of $witness(P,B)$. Therefore, A is the next vertex in the execution. Vertex C follows A in the execution as it is the sole successor of A. Since the edge that produced witness **t** ($P \rightarrow A$) has been traversed already, we read the next witness. Witness **u** is a member of $witness(C,P)$ but not $witness(C,EXIT)$, so vertex P follows C. At vertex P, witness **u** is still valid (since the edge $B \rightarrow A$ has not been traversed yet) and determines B as P's successor. Continuing in this manner, the original execution can be reconstructed.

If a witness w is a member of both $witness(G,\ Ewit,\ p,\ a)$ and $witness(G,\ Ewit,\ p,\ b)$, where $a \neq b$, then two *different* executions of G generate the same trace, which makes regeneration based solely on control-flow and trace information impossible. For example, in Figure 3.11, if the edge $P \rightarrow A$ did not generate a witness, then $witness(P,A)$ = **{ u, v, EOF }** and $witness(P,B)$ = **{ u, v }**. The executions (P, A, C, P, B, C, $EXIT$) and (P, B, C, $EXIT$) both generate the trace (**v, EOF**). This motivates our definition of the tracing problem:

DEFINITION. A set of edges, *Ewit*, *solves the tracing problem* for CFG G, denoted by $Trace(Ewit)$, *iff* for each predicate p in G with successors $q_1, ..., q_m$, for all pairs $(q_i,\ q_j)$ such that $i \neq j$, $witness(G,\ Ewit,\ p,\ q_i) \cap witness(G,\ Ewit,\ p,\ q_j) = \emptyset$.

It is straightforward to show that *Ewit* solves $Trace(Ewit)$ for CFG G *iff* $E - Ewit$ contains no diamonds or directed cycles. Optimally breaking diamonds and directed cycles is an NP-complete problem, as discussed in Section 3.1.4. Note that any solution to $Eprof(Ecnt)$ or $Vprof(Ecnt)$ is also a solution to $Trace(Ewit)$, as breaking all undirected cycles or all simple piped cycles is guaranteed to break all directed cycles and diamonds. Edges not in the maximum spanning tree of the CFG comprise *Ewit* and solve $Trace(Ewit)$ (but not necessarily optimally). However, for any CFG G in G^*, an optimal solution to $Eprof(Ecnt)$ is also an optimal solution to $Trace(Ewit)$ (because all directed cycles and diamonds are piped cycles and every cycle in a CFG from G^* is piped).

Given a CFG G, a set of edges *Ewit* that solves $Trace(Ewit)$, and the trace produced by an execution EX, the algorithm in Figure 3.12 regenerates the execution EX.

```
procedure regenerate(G: CFG; Ewit: set of witnessed edges; trace: file of witnesses )
declare
   pc, newpc : vertices;
   wit : witness;
begin
      pc := root-vertex(G); wit := read(trace);
      output(pc);
      do
          if not IsPredicate(pc) then
                 newpc := successor(G, pc);
          else
                 newpc := q such that wit ∈ witness(G, Ewit, pc, q)
          fi
          if pc→newpc ∈ Ewit then wit := read(trace) fi
          pc := newpc;
          output(pc);
      until ( pc = EXIT )
end
```

Figure 3.12. Algorithm for regenerating an execution from a trace.

3.2.2. Multi-Procedure Tracing

Unfortunately, tracing does not extend as easily to multiple procedures as profiling. There are several complications that we illustrate with the CFG in Figure 3.11. Suppose that basic block *B* contains a call to procedure *X* and execution proceeds from *P* to *B*, where procedure *X* is called. After *X* returns, suppose that *C* executes. This call creates problems for the regeneration process since the witnesses generated by procedure *X* and the procedures it invokes, possibly an enormous number of them, precede witness **v** in the trace file.

In order to determine which branch of predicate *P* to take, the witnesses generated by procedure *X* could be buffered or witness set information could be propagated across calls and returns (*i.e.*, along call graph edges as well as control-flow edges). The first solution is impractical since the number of witnesses that may have to be buffered is unbounded. The second solution is made expensive by the need to propagate information interprocedurally, and is complicated by multiple calls to the same procedure, calls to unknown procedures, and recursive calls. Furthermore, if witness numbers are reused in different procedures, which greatly reduces the amount of storage needed for a witness, then the second approach becomes even more complicated. (If a separate trace file were maintained for each procedure then all these problems would disappear and extending tracing to multiple procedures would be quite straightforward. However, this solution is not practical for anything but toy programs for obvious reasons.)

Our solution places "blocking" witnesses on some edges of the paths from a predicate to a call site, and from a predicate to the *EXIT* vertex. This ensures that whenever the regeneration procedure is in CFG G and reads a witness to determine which branch of a predicate to take, the witness will have been generated by an edge in G.[3]

DEFINITION. The set *Ewit* has the *blocking property* for CFG G iff there is no predicate p in G such that there is a witness-free directed path from p to the *EXIT* vertex or a vertex containing a call.

DEFINITION. The set $\{ Ewit_1, ..., Ewit_m \}$ solves the *tracing problem for a set of CFGs* $\{G_1, ..., G_m\}$ iff, for all i, $Ewit_i$ solves $Trace(Ewit_i)$ for G_i and $Ewit_i$ has the blocking property for G_i.

The regeneration algorithm in Figure 3.12 need only be modified to maintain a stack of currently active procedures. When the algorithm encounters a call vertex, it pushes the current CFG name and *pc* value onto the stack and starts executing the callee. When the algorithm encounters an *EXIT* vertex, it pops the stack and resumes executing the caller.

An easy way to ensure that *Ewit* has the blocking property is to include each incoming edge to a call or *EXIT* vertex in *Ewit*. Figure 3.13 illustrates why this approach is suboptimal. The shaded vertices (B, I, and H) are call vertices. In the first subgraph, a blocking witness is placed on each incoming edge to a call vertex (black dots). In addition, a witness is needed on edge $B \rightarrow D$ (white dot). This placement is suboptimal because the witness on edge $H \rightarrow I$ is not

---

[3]In some tracing applications, data other than witnesses (such as addresses) are also written to the trace file. Vertices in the CFG that generate addresses can be blocked with witnesses so that no address is ever mistakenly read as a witness. It would also be feasible in this situation to break the trace file into two files, one for the witnesses and the other for the addresses, to avoid placing more blocking witnesses.

needed, and because the witnesses on edges $B{\rightarrow}D$ and $G{\rightarrow}I$ (with cost = 3) can be replaced by witnesses on edges $B{\rightarrow}D$ and $B{\rightarrow}E$ (with cost = 2). In the second subgraph, blocking witnesses are placed as far from call vertices as possible, resulting in an optimal placement.

Consider a call vertex v and any directed path from a predicate p to v such that no vertex between p and v in the path is a predicate. For any weighting of G, placing a blocking witness on the outgoing edge of predicate p in each such path has cost equal to placing a blocking witness on each incoming edge to v (since no vertex between p and v is a predicate). However, placing blocking witnesses as far away as possible from v ensures that no blocking witnesses are redundant. Furthermore, placing the blocking witnesses in this fashion increases the likelihood that they solve $Trace(Ewit)$.

In general, it is not always the case that a blocking witness placement will solve $Trace(Ewit)$. Therefore, computing $Ewit$ becomes a two step process: (1) place the blocking witnesses; (2) ensure that $Trace(Ewit)$ is solved by adding edges to $Ewit$. The details of the algorithm follow:

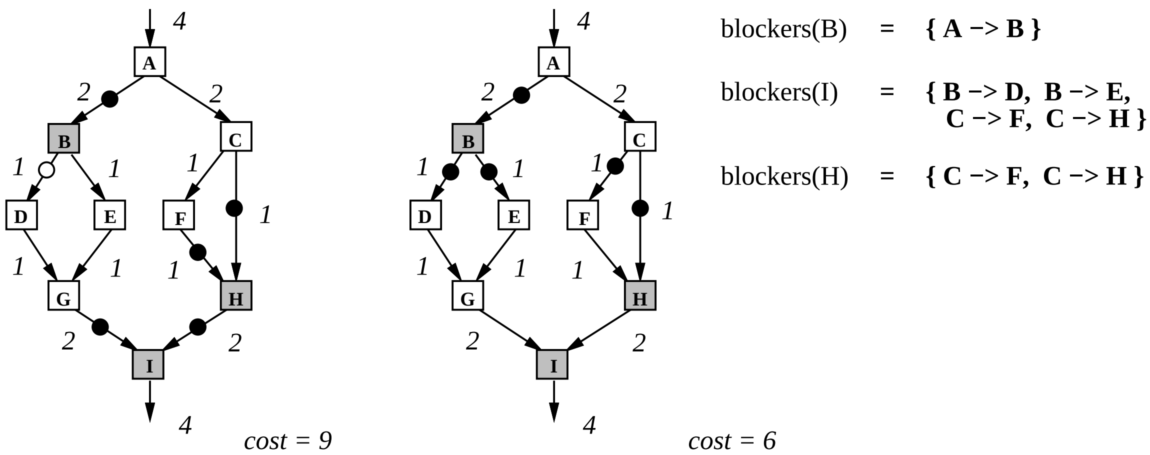


Figure 3.13. Two placements of blocking witnesses: a suboptimal placement and an optimal placement.

DEFINITION. Let v be a vertex in CFG G. The *blockers* of v are defined as follows:

$$\text{blockers}(G, v) = \{\, p \rightarrow x_0 \mid \text{there is a path } p \rightarrow x_0 \rightarrow \cdots \rightarrow x_n \text{ where } p \text{ is a predicate}, \; v = x_n, \text{ and for } 0 \le i < n,\ x_i \text{ is not a predicate} \,\}$$

First, for each vertex v that is a call or *EXIT* vertex, all edges in blockers(G, v) are added to *Ewit* (which is initially empty). To ensure that *Ewit* solves *Trace*(*Ewit*), we must add additional edges to *Ewit* so that $E - Ewit$ contains no diamonds or directed cycles. The maximum spanning tree algorithm can be modified to add these edges. No edge that is already in *Ewit* is allowed in the spanning tree.[4] Edges that are not in the spanning tree are added to *Ewit*, which guarantees that *Ewit* solves *Trace*(*Ewit*). Applying this algorithm to the control-flow fragment in Figure 3.14(a), the blocking phase adds the black dot edges to *Ewit*. The spanning tree phase adds the white dot edge to *Ewit*.

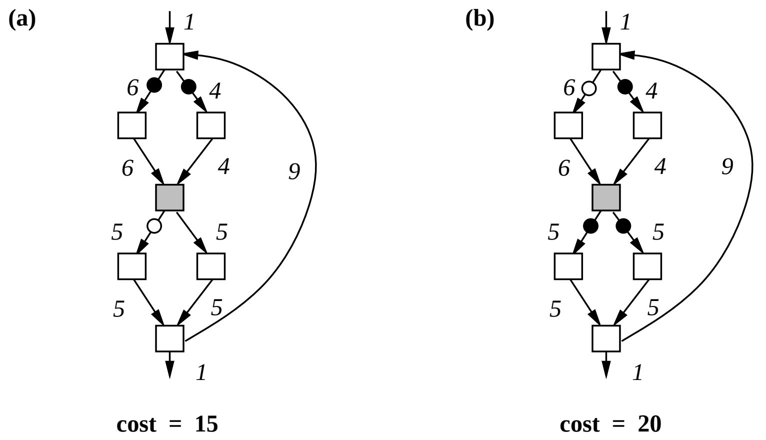


Figure 3.14. Ordering of blocking witness placement and spanning tree placement affects optimality.

[4]The modified spanning tree algorithm may not actually be able to create a spanning tree of G because of the edges already in *Ewit*. In this case the algorithm simply identifies the maximal cost set of edges in $E - Ewit$ that contains no (undirected) cycle.

One might question whether it is better to reverse the above process and first compute an *Ewit* that solves $Trace(Ewit)$, using the maximum spanning tree algorithm, and add blocking witnesses as needed afterwards. Figure 3.14(b) shows that this approach can yield undesirable results. The black dot edges are placed by the spanning tree phase and solve $Trace(Ewit)$ but do not satisfy the blocking property. The white dot edge must be added to satisfy the blocking property and creates a suboptimal *Ewit*.

3.3. A HEURISTIC WEIGHTING ALGORITHM

In order to profile or trace efficiently, instrumentation code should be placed in areas of low execution frequency. It may appear that to find areas of low execution frequency requires profiling. However, structural analysis of the CFG can often accurately predict that some portions are less frequently executed than others. This section presents a simple heuristic for weighting edges, based solely on control-flow information. As shown in Section 3.4, this simple heuristic is quite effective in reducing instrumentation overhead. The basic idea is to give edges that are more deeply nested in conditional control structures lower weight, as these areas will be less frequently executed. In general, every path through a loop requires instrumention. However, within a loop containing conditionals, we would still like instrumentation to be as deeply nested as possible. For the CFG in Figure 3.15, the heuristic will generate the weighting shown in case (a). Any weighting of a CFG (*i.e.*, edge frequencies satisfying Kirchoff's flow law) that assigns each edge a non-zero weight will give edges that are more deeply nested lower weight. As discussed in Section 3.5, there are expensive matrix-oriented methods for generating weightings. Our heuristic has the advantage that it requires only a depth-first search and topological traversal of the CFG.

The heuristic has several steps. First, a depth-first search of the CFG from its root vertex identifies backedges in the CFG. The heuristic uses a topological traversal of the backedge-free graph of the CFG to compute the weighting. The weighting algorithm uses natural loops to identify loops and loop-exit edges [1]. The natural loop of a backedge $x \rightarrow y$ is defined as follows:

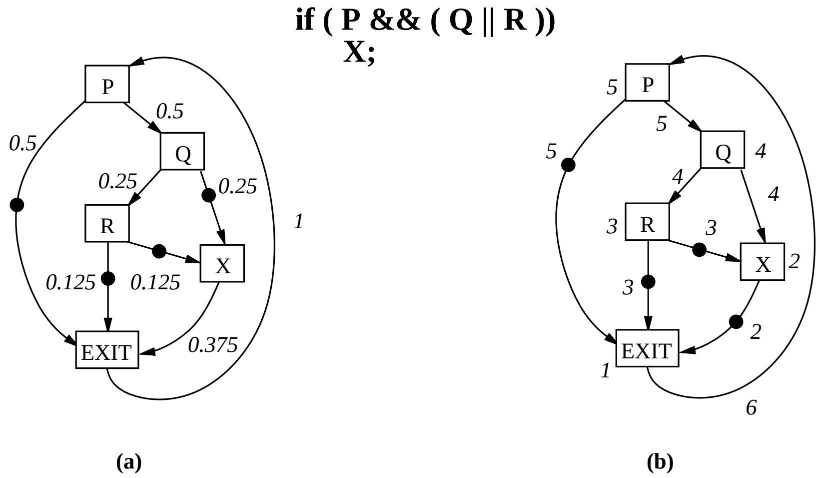


Figure 3.15. A program fragment, (a) its CFG with a weighting satisfying Kirchoff's flow, and an optimal edge counter placement (black dots). Case(b) shows a weighting derived using a post-order numbering of vertices (an edge's value is the post-order number of its source vertex), and the sub-optimal placement that results from finding a maximum spanning tree with respect to this weighting.

$$nat\text{-}loop\,(x \rightarrow y) =$$
$$\{y\} \cup \{\, w \mid \text{there is a directed path from } w \text{ to } x \text{ that does not include } y \,\}$$

A vertex is a loop-entry if it is the target of one or more backedges. The natural loop of a loop-entry y, denoted by $nat\text{-}loop\,(y)$, is simply the union of all natural loops $nat\text{-}loop\,(x \rightarrow y)$, where $x \rightarrow y$ is a backedge. If a and b are different loop-entry vertices, then either $nat\text{-}loop\,(a)$ and $nat\text{-}loop\,(b)$ are disjoint or one is entirely contained within the other. This nesting property is used to define the exit edges of a loop with entry y:

$$exit\text{-}edges\,(y) = \{\, a \rightarrow b \in E \mid a \in nat\text{-}loop\,(y) \text{ and } b \notin nat\text{-}loop\,(y) \,\}$$

Edge $a \rightarrow b$ is an exit edge if there exists a loop-entry y such that $a \rightarrow b \in exit\text{-}edges\,(y)$.

The heuristic assumes each loop iterates LOOP_MULTIPLIER times (for our implementation, 10 times) and that each branch of a predicate is equally likely to be chosen. Exit edges are specially handled, as described below. The weight of the edge $EXIT \rightarrow root$ is fixed at 1 and does not change. The edge $EXIT \rightarrow root$ is not treated as a backedge even though it is identified as such by depth-first search. The following rules describe how to compute edge and vertex weights:

(1) The weight of a vertex is the sum of the weights of its incoming edges that are not backedges.

(2) If vertex v is a loop-entry with weight W and $N = |exit-edges(v)|$, then each edge in $exit-edges(v)$ has weight W/N.

(3) If v is a loop-entry vertex then let W be the weight of vertex v times *LOOP_MULTIPLIER*, otherwise let W be the weight of vertex v. If W_E is the sum of the weights of the outgoing edges of v that are exit-edges, then each non-exit outgoing edge of v has weight $(W - W_E)/N$, where N is the number of non-exit outgoing edges of v.

The rules are applied in a single topological traversal of the backedge-free graph of a CFG (however, backedges are given weights by rule (3)). An edge is assigned a weight by the first rule that applies to it in the traversal, as follows. When vertex v is first visited during the traversal, the weights of its incoming non-backedges are known. Rule (1) determines the weight of vertex v. If vertex v is a loop-entry then rule (2) is used to assign a weight to each edge in $exit-edges(v)$. Finally, rule (3) determines the weight of each outgoing edge of v that is not an exit edge.

3.4. PERFORMANCE RESULTS

This section describes several experiments that demonstrate that the algorithms presented above significantly reduce the cost of profiling and tracing real programs. Sections 3.4.1 and 3.4.2 discuss the performance of the profiling and tracing algorithms, respectively. Section 3.4.3 considers some optimizations that can further decrease the overhead of profiling and tracing. Section 3.4.4 examines the effectiveness of the heuristic weighting algorithm.

3.4.1. Profiling Performance

We implemented the profiling counter placement algorithm in *qpt* [45], which is a basic block profiler similar to MIPS's *pixie* [67]. *Qpt* instruments object code and can either insert counters in every basic block in a program (*redundant* mode) or along the subset of edges identified by the spanning tree algorithm (*optimal* mode).

We used the SPEC benchmark suite to test *qpt* [14]. This is a collection of 10 moderately large Fortran and C programs that is widely used to evaluate computer system performance. The programs were compiled at a high level of optimization (either -O2 or -O3, which does interprocedural register allocation). However, we did not use the MIPS utility *cord*, which reorganizes blocks to improve cache behavior, or interprocedural delay slot filling. Both optimizations confuse a program's structure and greatly complicate constructing a control-flow graph. Timings were run on a DECstation 5000/200 with local disks and 96MB of main memory. Times are elapsed times.

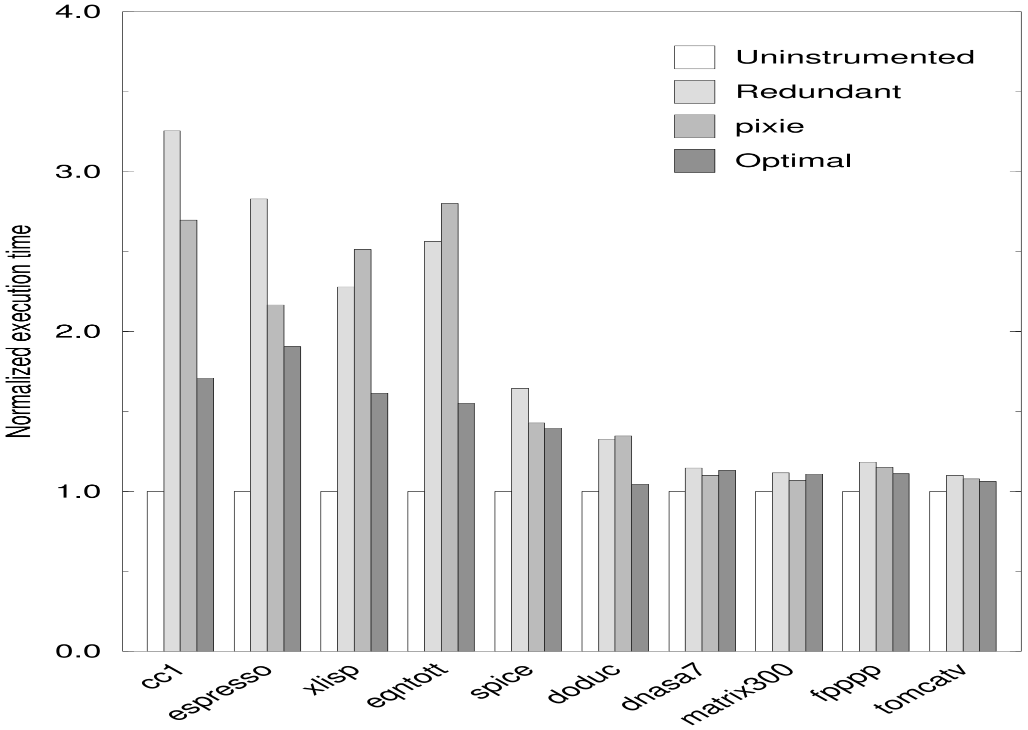
Table 3.1 describes the 10 benchmarks and shows the size of the object files and the time required to insert profiling code in redundant and optimal mode (keep in mind that *qpt* has not been tuned because its current speed is more than adequate for most executables encountered in practice). As can be seen, instrumenting for optimal profiling is slightly (22-38%) slower than instrumenting for redundant profiling. This is due to the extra work to find the loops in a CFG and to compute a weighting, to drive the maximum spanning tree algorithm. In practice, this extra instrumentation overhead is quickly regained from the reduction in profiling overhead.

SPEC Benchmark	Description	**Size** (bytes)	**Redundant** (sec.)	**Optimal** (sec.)	**Increase** (Opt./Red.)
cc1 (C)	C compiler	1075840	9.2	12.4	1.35
espresso (C)	PLA minimization	298032	2.2	2.9	1.32
xlisp (C)	Lisp interpreter	175920	3.8	4.8	1.27
eqntott (C)	Boolean eqns. to truth table	94924	1.9	2.5	1.32
spice	Circuit simulation	551836	1.1	1.4	1.27
doduc	Monte Carlo hydrocode simul.	280940	1.9	2.5	1.32
dnasa7	Floating point kernels	162996	1.1	1.4	1.27
matrix300	Matrix multiply	122440	0.9	1.1	1.22
fpppp	Two-electron integral deriv.	254720	1.7	2.1	1.24
tomcatv	Vectorized mesh generation	125316	0.8	1.1	1.38

Table 3.1. SPEC benchmarks. Size of input object files and times for instrumenting programs for Redundant and Optimal profiling. The first four programs are C programs. The remainder are FORTRAN programs.

Graph 3.1 shows the (normalized) execution time of the benchmarks without profiling, with *qpt* redundant profiling, with *pixie* profiling (which inserts a counter in each basic block), and with *qpt* optimal profiling. *Pixie* rewrites the program to free 3 registers, which enables it to insert a code sequence that is almost half the size of the one used by *qpt* (6 instructions vs. 11 instructions). Of course, *pixie* may have to insert spill code in order to free registers.

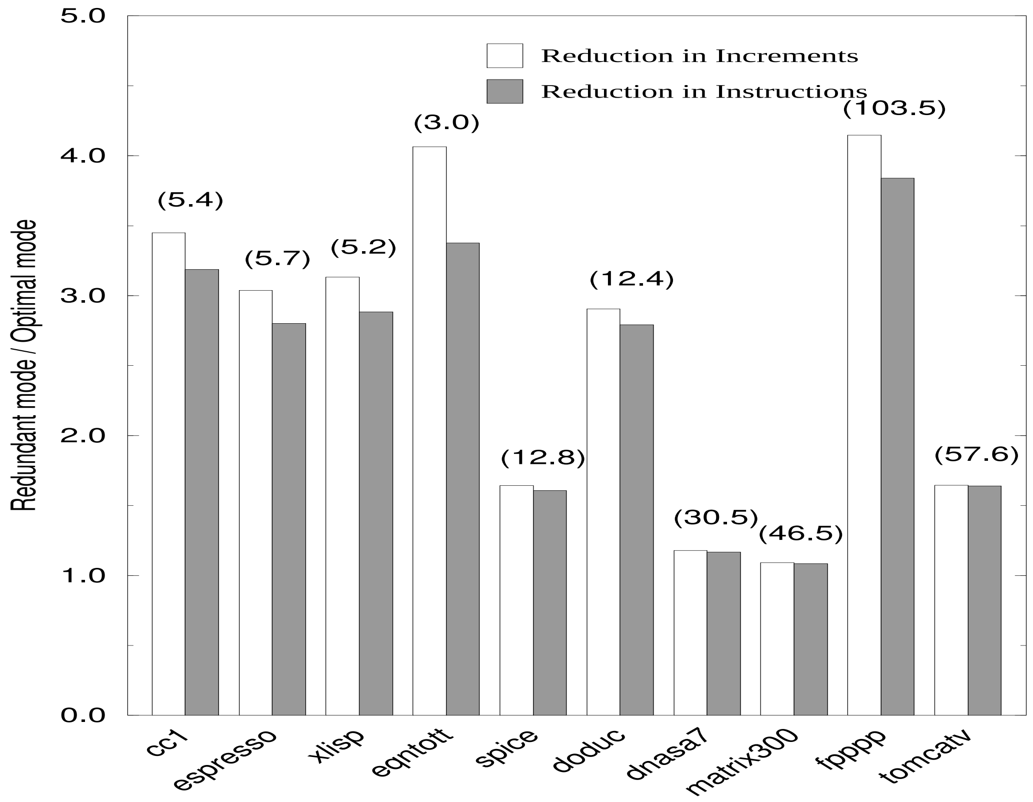
As can be seen from Graph 3.1, Optimal profiling reduces the overhead of profiling dramatically over Redundant profiling, from 10-225% to 5-91%. These timings are affected by variations in instruction and data cache behavior caused by instrumentation. We measured profiling improvement in another way that factors out these variations. Graph 3.2 records the reduction in counter increments in going from Redundant to Optimal profiling (*i.e.*, the number of counter



Graph 3.1. Normalized profiling execution times. For Redundant profiling, *qpt* inserts a counter in each basic block (vertex). For Optimal profiling, *qpt* inserts a counter along selected edges (*Eprof* (*Ecnt*)). *Pixie* is a MIPS utility that inserts a counter in each basic block.

increments in Redundant mode / the number of increments in Optimal mode). The graph also records the reduction in number of instrumentation instructions executed (assuming 5 instructions for a counter increment and 1 instruction for an unconditional branch for the edge profiling code). In general, this reduction is less than the reduction in counter increments since edge profiling may require the insertion of unconditional jumps.

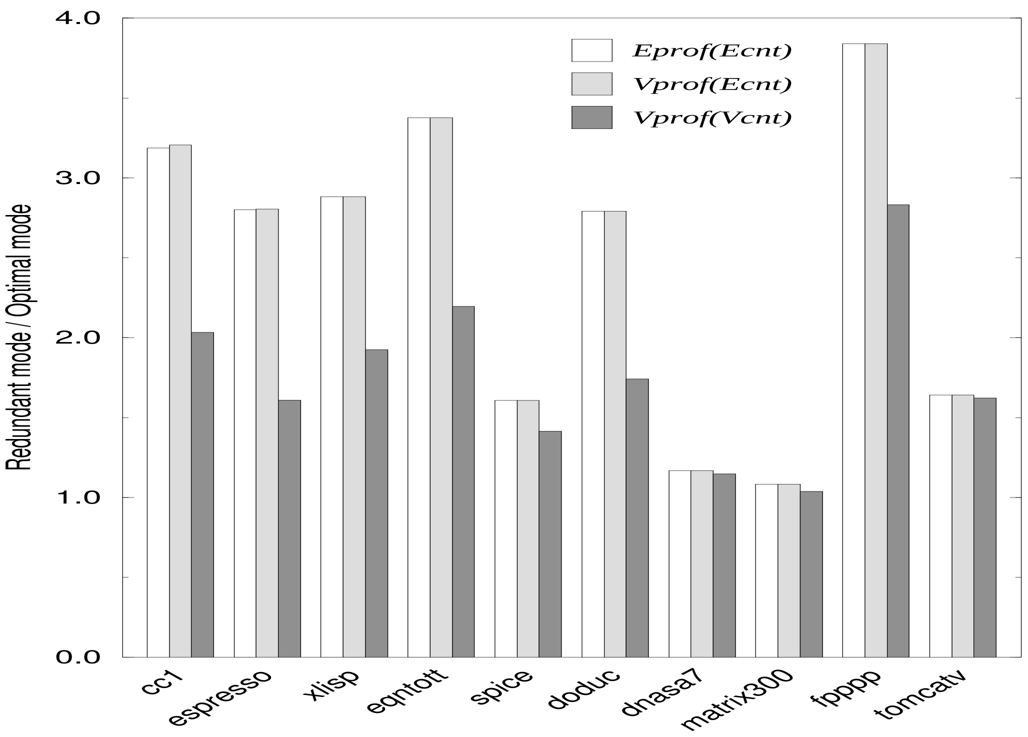
Fortunately, the greatest improvements occurred in programs in which profiling overhead was largest, since these programs had more conditional branches and more opportunities for



Graph 3.2. Reduction in counter increments and instrumentation instructions due to optimized counter placement, as guided by the heuristic weighting described in Section 3.3. Reduction in increments is (number of counter increments for Redundant profiling / number of counter increments for Optimal profiling). Reduction in instrumentation is (5 * number of basic blocks) / (5 * increments + number of extra jumps). The average dynamic basic block size (in instructions) for each program is shown in parenthesis.

optimization. For programs that frequently executed conditional branches, the improvements were large. For the four C programs (*cc1*, *espresso*, *xlisp*, and *eqntott*), the placement algorithm reduced the number of increments by a factor of 3-4 and the overhead by a factor of 2-3. For the Fortran programs, the improvements varied. In programs with large basic blocks that execute few conditional branches (where profiling was already inexpensive), improved counter placement did not have much of an effect on the number of increments or the cost of profiling. The FORTRAN program *doduc*, while it has a dynamic block size of 12.4 instructions, has “an abundance of short branches” [14] that accounts for its reduction in counter increments. The decrease in run time overhead for *doduc* was substantial (38% to 5%). The *fpppp* benchmark produced an interesting result. While it showed the largest reduction in counter increments, the overhead for measuring every basic block was already quite low at 18% and the average dynamic basic block size was 103.5 instructions. This implies that large basic blocks dominated its execution. Thus, even though many basic blocks of smaller size executed (yielding the dramatic reduction in counter increments), they contributed little to the running time of the program.

Graph 3.3 compares the reduction in dynamic instrumentation overhead for the *Eprof* (*Ecnt*) algorithm (optimal profiling), the *Vprof* (*Ecnt*) heuristic, and Knuth and Stevenson’s *Vprof* (*Vcnt*) algorithm, as compared to redundant profiling (measure at every vertex). All algorithms used the same weighting to compute a counter placement. Given a counter placement for one of the algorithms, we used the profile information collected from a previous run to determine how many times each counter would have been incremented and how many extra jumps would have been needed (*Vprof* (*Vcnt*) does not require extra jumps since counting code is placed on vertices). By doing so, we avoided instrumenting and running the programs for every algorithm, while still collecting accurate results. For all the benchmarks, *Eprof* (*Ecnt*) is superior to *Vprof* (*Vcnt*), producing a greater reduction in instrumentation instructions, as predicted. The heuristic for *Vprof* (*Ecnt*) yields almost no improvement over *Eprof* (*Ecnt*), as there are very few cases when a counter can be eliminated.



Graph 3.3. Comparison of instrumentation reduction of *Eprof* (*Ecnt*), *Vprof* (*Ecnt*), and *Vprof* (*Vcnt*). The larger a plot, the better (*i.e.*, the greater the reduction of instrumentation code). Instrumentation reduction = (5 * basic blocks) / (5 * increments + number of extra jumps).

Table 3.2 provides statistics on the number of edges in each program ("Total Edges"), the number of edges that had counting code placed on them using the spanning tree algorithm ("Profiled Edges"), and the number of profiled edges that did not require the insertion of an unconditional jump ("No-Jump Edges"). We make two observations. First, notice that the percentage of all edges that are profiled is in the narrow range of 39-46%. This is consistent with the facts that most CFGs have almost (but not quite) twice as many edges as vertices and that the number of edge counters required for edge profiling is $|E| - (|V| - 1)$. Second, less than half of all profiled edges require the insertion of an unconditional jump.

Program	Total Edges	Profiled Edges #	Profiled Edges % of Total	No-Jump Edges #	No-Jump Edges % of Profiled
cc1	48398	20577	43	10533	51
espresso	11059	4540	41	2426	53
xlisp	4813	2207	46	1264	57
eqntott	3095	1296	42	756	58
spice	15145	5888	39	3131	53
doduc	7957	3128	39	1672	53
dnasa7	5517	2274	41	1241	55
matrix300	4744	1969	42	1116	57
fpppp	7042	2887	41	1630	56
tomcatv	4661	1923	41	1099	57

Table 3.2. Static statistics on control-flow edges. “Total Edges” shows the total number of control-flow graph edges in each program. “Profiled Edges” shows the number of edges that had counters placed on them using the spanning tree algorithm. “No-Jump Edges” shows the number of profiled edges that do not require the insertion of an unconditional jump.

3.4.2. Tracing Performance

The witness placement algorithm was implemented in the AE program-tracing system [44], which has since been incorporated as part of the *qpt* tool. AE originally recorded the outcome of each conditional branch and used this record to regenerate a full control-flow trace. One complication is that AE traces both the instruction and data references so a trace file contains information to reconstruct data addresses as well as the witnesses. Combining this information in one file requires additional blocking witnesses, as described in Section 3.2.2.

Table 3.3 shows the reduction in total file size (“File”), witness trace size (“Trace”), and execution time that result from switching the original algorithm of recording each conditional (“Old”)

Program	**Old File** (bytes)	**New File** (bytes)	**Old/ New**	**Old Trace** (bytes)	**New Trace** (bytes)	**Old/ New**	**Old Run** (sec.)	**New Run** (sec.)	**Old/ New**
compress	6,026,198	4,691,816	1.3	2,760,522	926,180	3.0	6.6	5.4	1.2
sgefa	1,717,923	1,550,131	1.1	1,298,882	1,131,091	1.2	4.1	4.5	0.9
polyd	19,509,062	16,033,055	1.2	5,523,958	2,047,951	2.7	19.0	15.5	1.2
pdp	11,314,225	10,875,475	1.0	1,496,013	1,057,263	1.4	10.4	9.2	1.1

Table 3.3. Improvement in the AE program-tracing system. Old refers to the original version of AE, which recorded the outcome of every conditional branch. New refers to the improved version of AE, which uses the witness placement algorithm of Section 3.2. File refers to the total size of the recorded information, which includes both witness and data references. Trace refers to the total size of the witness information.

to the witness placement described in Section 3.2 (“New”). As with the profiling results, the programs with regular control-flow, *sgefa* and *pdp*, do not gain much from the tracing algorithm. For the programs with more complex control-flow, *compress* and *polyd*, the tracing algorithm reduced the size of the trace file by factors of 3 and 2.7 times, respectively.

In the discussion of tracing we assumed that a standard representation was used for witnesses (per CFG). In modern architectures it is convenient for this representation to be a multiple of a byte. Thus, it is often the case that we record more bits per witness than necessary. We explored another method for tracing, called *bit tracing*, which seeks to reduce the size of the trace. With bit tracing, each outgoing edge of a predicate vertex generates a witness and witness values are reused. For predicates with two successors, only one bit of information is required to distinguish its witness sets. In general, a predicate with N successors requires $\log_2 N$ bits. Figure 3.16 illustrates the tradeoff between the spanning tree approach and bit tracing. In case (a), witnesses are placed according to the spanning tree approach. No pair of distinct witnesses from the set **{ a, b, c, d }** can be assigned the same value, so two bits per witness are required. In case (b), only one bit per witness is required. Any iteration of the loop in this CFG will generate three bits of trace. However, in case (a) the amount of trace generated per iteration can either be two or four bits. In this example, neither witness placement is a clear winner.

If compared to the spanning tree approach that naively uses a byte (or more, if needed) of storage per witness, bit tracing is clearly superior. Although more instrumentation code is executed, less trace is generated, which reduces I/O overhead. This method decreases the size of the trace 3-7 times over the spanning tree approach. However, as shown in Figure 3.16, by using only as many bits as necessary, the spanning tree approach can be improved upon. In this example, two bits per witness are needed. In general, if there are N witnesses for a CFG then at most $\log_2 N$ bits per witness are needed. However, there are situations where witness values can be reused, possibly decreasing the number of bits needed. This is complicated by the fact that different placements of witnesses may give rise to different opportunities for the reuse of values. Further investigation in optimizing the spanning tree approach is clearly needed.

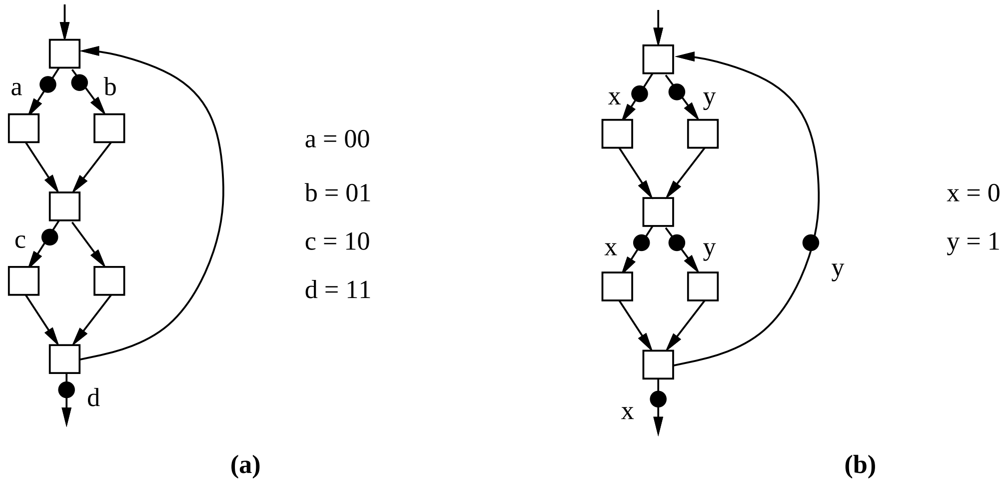


Figure 3.16. Tradeoff between (a) placing witnesses according to the spanning tree approach and (b) placing witnesses on every outgoing edge of a predicate vertex.

Bit tracing avoids the multi-procedure tracing problem discussed in Section 3.2.2 as there is no witness-free directed path from a predicate to a call vertex. If an address trace also is generated from the program, bit tracing requires that two separate files be maintained (for efficiency), one for the instruction trace and one for the address trace. The cost of bit tracing is the additional implementation complexity required to manage witnesses at the bit level.

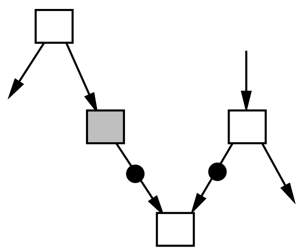
3.4.3. Optimizations

Several optimizations can further decrease the overhead of profiling and tracing. The first optimization, *register scavenging*, is specific to instrumenting object code. For RISC machines, counter increment code requires two registers, one to hold the counter's address (because addressing on RISC machines is done by indirection off of a register) and one to hold the counter's value. If both registers need to be saved and restored (to preserve their values), the instrumentation code jumps from 5 to 11 instructions. Register scavenging notes the unused caller-saved registers in a procedure. These registers can be used by instrumentation code without preserving their values, since the procedure's callers expect these registers to be modified.[5] For many of the benchmarks, specifically the FORTRAN programs with large basic blocks and few unused registers, register scavenging had little effect on execution overhead. For other benchmarks, the results varied from small reductions of a few percent to larger reductions in the range of 6-21%.

The second optimization can substantially reduce profiling overhead by removing counters from loops. If the number of iterations of a loop can be determined before the loop executes or from an induction variable whose value is recorded before and after the loop, then a counter can be eliminated from the loop body (allowing one counter-free path through the loop). Both Sarkar and Goldberg have successfully implemented this approach in profiling tools [24, 64]. For example, Goldberg reports that for *eqntott* the reduction in increments increased from 4.3 to 7.7 after adding induction variable analysis. Some scientific codes benefitted greatly from this analysis (a 33-fold decrease in instrumentation code executed for *matrix300*). However, for pointer-chasing programs such as *xlisp* the benefits of this analysis were quite small, as few induction variables are present in such programs.

As mentioned before, placing instrumentation code on edges may require the insertion of jumps in order to avoid executing other instrumentation code. For example, in the control-flow fragment of Figure 3.17(a) there are two instrumented incoming edges to a vertex. Because we

(a)



(b)

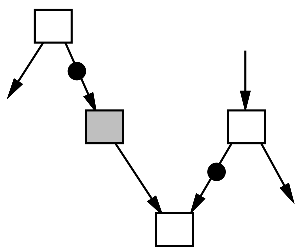


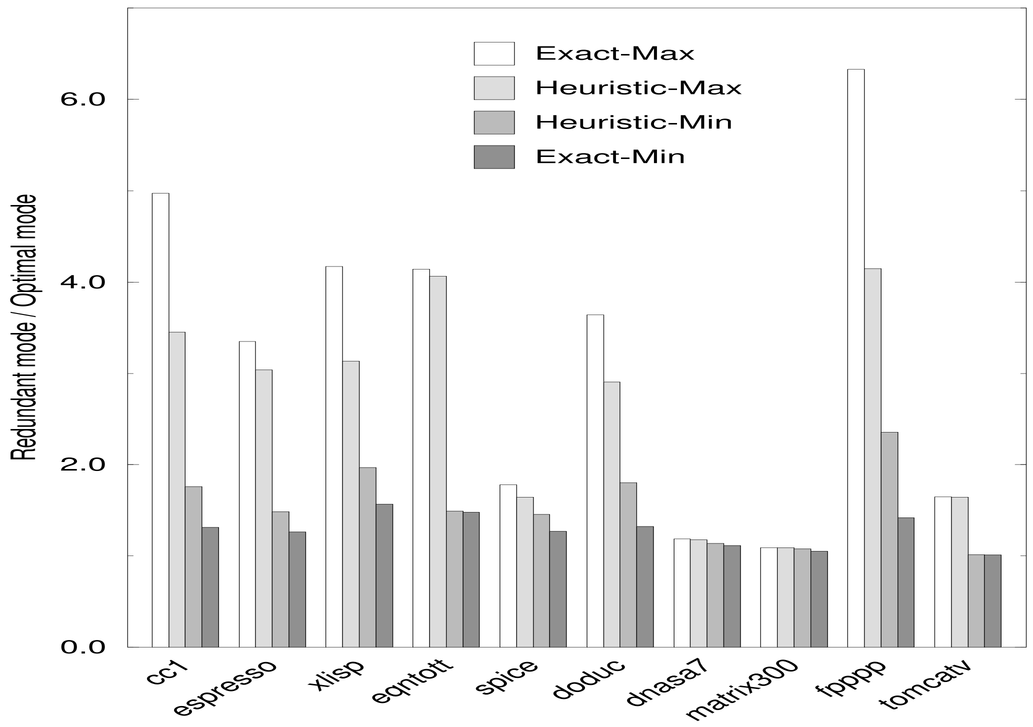
Figure 3.17. (a) Placement requiring insertion of jump. (b) No jumps required.

[5]We discuss the problems of register scavenging and instrumenting object files in greater detail elsewhere [46].

use the general rule that the instrumentation code associated with an edge is placed just before the code associated with the vertex that is the target of the edge, this fragment will require at least one unconditional jump (in order to jump over the instrumention code associated with the other edge). However, the grey vertex has only one incoming edge and only one outgoing edge, so the instrumentation point can be moved from its outgoing edge to its incoming edge, resulting in the placement in case (b). This placement may require no extra jumps (unless the grey vertex's outgoing edge is a fall-through). Jump optimization searches for vertices with one incoming and one outgoing edge with instrumentation code on the outgoing edge. The instrumentation code is simply moved to the incoming edge. This simple optimization may reduce (and will never increase) the number of extra jumps. In the case of *xlisp*, this optimization reduced execution overhead by 10 percent.

3.4.4. Effectiveness of the Heuristic Weighting Algorithm

The effectiveness of the heuristic weighting algorithm was measured in two ways, as presented in Graph 3.4. First, we measured the reduction in counter increments (number of increments in Redundant mode / numer of increments in Optimal mode) using an exact edge weighting from a previous run of the same program with identical input. This number, "Exact-Max", represents the best one could hope to do without semantics-based optimizations (such as induction variable analysis). Second, for both the heuristic and exact weightings, we also computed what the reduction in increments would be if a minimum spanning tree were used to place counters. While the maximum spanning tree places counters in less frequently executed areas of the CFG, a minimum spanning tree places counters in more frequently executed areas. Thus, "Exact-Min" is the worst possible reduction for the spanning tree algorithm. As the difference between "Exact-Min" and "Exact-Max" shows, there is great variation in the reduction in counter increments, depending on which spanning tree is chosen. The heuristic is clearly successful at predicting areas of low execution frequency, as there is a noticeable difference between the reduction in increments for the counter placements determined by maximum and minimum spanning trees. Note that "Heuristic-Max" always produced a better reduction than "Heuristic-Min". The difference in

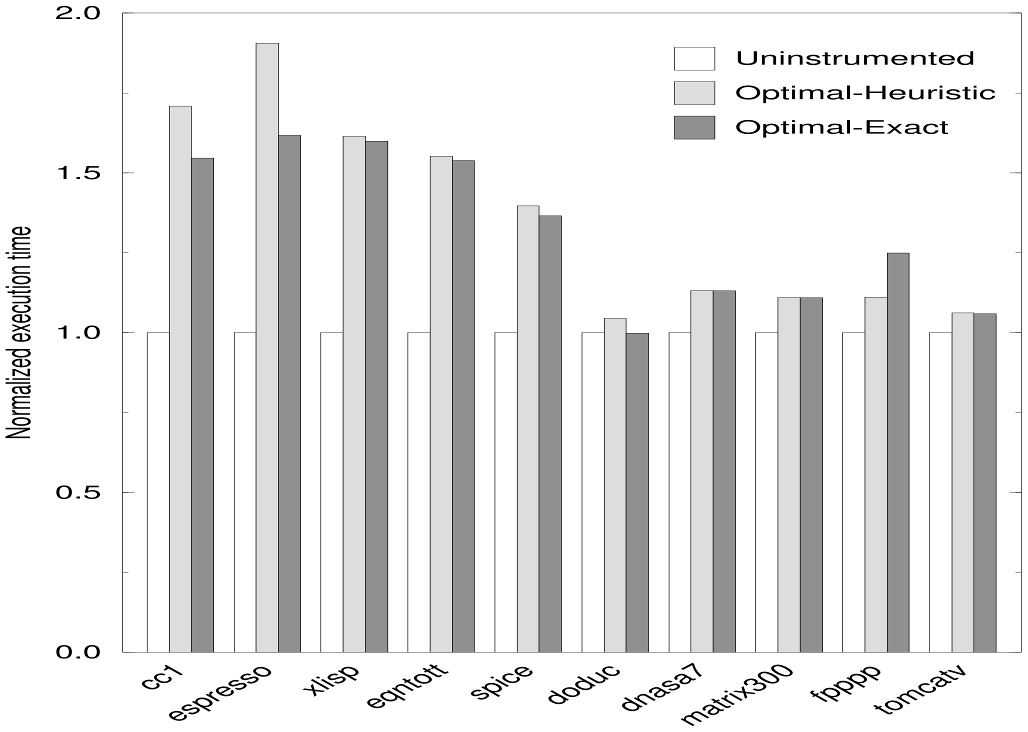


Graph 3.4. Heuristic weighting vs. exact weighting: reduction in increments (Redundant/Optimal). Reductions in increments are computed for counter placements from both maximum and minimum spanning trees.

reduction between the heuristic and exact weightings was usually small (ranging from 1% to 34%). Not surprisingly, the heuristic was quite accurate for the FORTRAN programs with few conditional branchs.

Graph 3.5 shows the normalized times for the benchmarks run under Optimal profiling for the heuristic weighting (corresponds to “Heuristic-Max” in Graph 3.4) and exact weighting (corresponds to “Exact-Max” in Graph 3.4). In one case (*fpppp*), the run time with the exact weighting is greater than the run time with the heuristic weighting. Such an aberration is most likely due to different instruction and data cache behavior of the instrumented program under the different counter placements, and requires further investigation.

The heuristic weighting algorithm assumes each branch of a predicate is equally likely to be chosen. For most programs, varying this probability does not have a great effect on instrumentation overhead. However, weighting schemes that attempt to pick likely branch directions



Graph 3.5. Normalized profiling times for heuristic and exact weightings.

independently may have greater success. For example, favoring edges leading to blocks containing loops (which have a high dynamic cost) reduces instrumentation overhead for a few of the benchmarks.

3.5. RELATED WORK

3.5.1. Edge Profiling

The spanning tree solution to *Eprof* (*Ecnt*) has been known for a long time. In the area of network programming, the problem is known as the specialization of the simplex method to the network program [37]. Knuth describes how to use the spanning tree for profiling in [40]. Other authors that have written about the application of the spanning tree to profiling include Goldberg [24], Samples [63], and Probert [55]. As far as we know, Goldberg and Samples are the only other researchers that have implemented the spanning tree approach and performed significant experimentation with real programs. Their work occurred concurrently with ours.

Goldberg implemented edge profiling by instrumenting executable files [24]. His profiler was built as part of a system to analyze the memory performance of programs [25]. Goldberg optimized his instrumentation in two ways that we do not consider. First, his tool selected the two statically least-used registers in the executable and eliminated all uses by inserting loads and stores around existing uses of these registers. This allows every counting code sequence to use these registers without saving and restoring them. A similar approach is used by MIPS's *pixie* profiling tool [67]. As a result, the number of instructions needed to increment a counter in memory can be cut roughly in half. Our tool only looks for free registers to scavenge and often must save and restore registers in the counter increment code sequence. Second, Goldberg identifies simple loop induction variables. This allows a counter to be eliminated from a loop (because the number of iterations can be inferred from the beginning and ending values of the induction variable), lowering instrumentation overhead drastically for scientific codes. Our tool does not perform this optimization.

Samples considers a refinement that takes into account the unconditional jump that may have to be inserted into the profiled program when placing a counter on an edge. His algorithm places counters on a mixture of edges and vertices to reduce the number of unconditional jumps as well as the number of counter increments. His approach is useful for architectures in which the cost of an unconditional jump is comparable to the cost of incrementing a counter in memory. However, as mentioned before, Samples' results show that the overhead incurred by mixed placements did not differ much from edge placements.

Probert discusses solving *Eprof* (*Vcnt*), which is not always possible in general. Using graph grammars, he characterizes a set of "well-delimited" programs for which *Eprof* (*Vcnt*) can always be solved. This class of graphs arises by introducing "delimiter" vertices into well-structured programs. Probert discusses how to find a minimal number of vertex measurement points as opposed to a minimal cost set of measurement points.

Sarkar describes how to choose profiling points using control dependence and has implemented a profiling tool for the PTRAN system [64]. His algorithm finds a minimum sized solution to *Eprof* (*Ecnt*) based on a variety of rules about control dependence, as opposed to the spanning tree approach. There are several other major differences between his work and our work: (1) His algorithm only works for a subclass of reducible CFGs; (2) His algorithm does not use a weighting to place counters at points of lower execution frequency. As a result, the algorithm may produce suboptimal solutions; (3) When the bounds of a **DO** loop are known before execution of the loop, his algorithm eliminates the loop iteration counter, as done by Goldberg.

3.5.2. Vertex Profiling

Knuth and Stevenson exactly characterize when a set of vertices *Vcnt* solves *Vprof* (*Vcnt*) and show how to efficiently compute a minimum size *Vcnt* that solves *Vprof* (*Vcnt*) [41]. The authors note that their algorithm can be modified to compute a minimum cost solution to *Vprof* (*Vcnt*) given a set of measured or estimated vertex frequencies. Our work shows that it is less costly to measure vertex frequency by instrumenting edges rather than vertices.

3.5.3. Tracing

Ramamoorthy, Kim, and Chen consider how to instrument a single-procedure program with a minimal number of monitors, so the traversal of any directed path through the program may be ascertained after an execution [57]. This is equivalent to the tracing problem for single-procedure programs discussed here. The authors do not give an algorithm for reconstructing an execution from a trace or consider how to trace multi-procedure programs. Further, they are interested in finding a minimal *size* solution to the tracing problem, an NP-complete problem [47]. However, a minimum size solution does not necessarily yield a minimum cost solution.

3.5.4. Minimizing instrumentation overhead

A CFG has many spanning trees, each of which induces a counter placements with an associated run-time overhead cost. Section 3.3 presented our heuristic for estimating edge frequency in order to drive the maximum spanning tree algorithm. This section compares our heuristic to other methods for minimizing instrumentation overhead (which may include methods for estimating frequency). We use the CFG in Figure 3.15(a) as a basis for comparing the various heuristics discussed below. The weighting of this CFG satisfies the flow law and the edges with black dots are an optimal edge counter placement for profiling (with respect to this weighting). The other edges form a maximum spanning tree. As mentioned before, our heuristic will generate the weighting in case (a).

Forman discusses the problem of minimizing counter overhead with the spanning tree approach from a graph theoretic perspective [20]. He defines a partial order on the spanning trees of a CFG such that for any weighting, if a spanning tree T is not a least element in the partial order then there is some spanning tree lower in the order that induces a counter placement with lower cost than the one induced by T. Of course, there may be more than one least element in the partial order. The spanning tree in Figure 3.15(a) is a least element. Forman proposes a structural method for computing a least element, but it works only for structured CFGs. Our heuristic works for any CFG. He also proposes a more general solution that generates a weighting, given branch probabilities for the predicate vertices in any CFG. A maximum spanning tree found under this weighting corresponds to a least element in Forman's partial order. To generate the weighting requires matrix operations on what are essentially adjacency matrix representations of the CFG. As such, this general approach would be much slower than our heuristic, which operates directly on the control-flow graph structure. Our heuristic generates edge frequencies satisfying the flow law and can easily be adapted to take branch probabilities into account.

Goldberg developed a heuristic for his profiling tool that uses a post-order numbering of the vertices in the CFG (as determined by a depth-first search from the *root* vertex) to assign edge weights [24]. He defines an edge's weight to be the post-order number of its source vertex. However, if an edge is a loop backedge then it is given a weight larger than the number of vertices in the graph. The rationale for this heuristic is that "...a node always executes at least as many times as any of its descendants [successors]; hence, it seems best to place counters on nodes as far from the root as possible." This heuristic clearly does not produce a weighting satisfying the flow law, as Figure 3.15(b) shows. Because the distance of an edge from the root vertex does not always correspond to its level of nesting, Goldberg's heuristic will not always lead to the best counter placements. In the example of Figure 3.15(b), the maximum spanning tree for the given weighting (determined by his heuristic) induces a sub-optimal counter placement.

Wall experimented with a number of heuristics for estimating basic block and procedure profiles solely from program text, reporting poor results [69]. Wall's heuristics use information about loop nesting and call graph structure to predict basic block and procedure profiles, but do not take into account conditional control-flow (*i.e.*, predicting that code that is more deeply nested in conditionals is executed less frequently), as our heuristic does. It is this aspect of our heuristic that is key to reducing instrumentation overhead (this is also the main idea behind Forman's partial order). With Wall's heuristic, every basic block that is contained in the same number of loops gets the same weight. In the example graph of Figure 3.15, each block would get the same weight, which is clearly not useful for the purposes of minimizing instrumentation cost.

Other authors have presented heuristics that are similar to ours, usually for the purpose of aiding code optimization. For example, Fisher, Ellis, Ruttenberg, and Nicolau use loop nesting level and programmer-supplied hints to estimate block execution frequency for trace scheduling [19]. However, few of these heuristics have the goal of producing edge frequencies satisfying the flow law.

None of the heuristics mentioned above nor our heuristic attempts to predict branch directions. If branches can be accurately predicted, then instrumentation code can be placed on the less frequently executed branch when a choice is possible. More recent work on branch prediction by the author of this thesis, in association with J. R. Larus, could be used in this application [5].

Chapter 4

EFFICIENTLY COUNTING PROGRAM EVENTS WITH SUPPORT FOR ON-LINE QUERIES

Many applications require counting how many times certain events occur in a program's execution. For example, instruction counts can be used to determine how much time is spent in a procedure [26]. Event counting can be used to implement countdown timers for debugger breakpoints or execution-driven simulators so that control returns from the executing program to the debugger or simulator after a certain number of events [49, 59]. Counts of synchronization events, I/O events, and system calls also can be used to measure the performance of parallel programs [27]. Furthermore, many of these applications require the capability to query the event count on-line, while the program executes, rather than off-line, after the program has terminated. For example, interactive performance measurement tools need to make such queries in order to update displays in a timely fashion.

This chapter investigates how to count events in a program execution efficiently, with support for on-line queries of the event count. We present a new method for efficiently counting and querying program events that uses program instrumentation. Rather than instrument every basic block in the program, our algorithms find select points in a program to instrument while guaranteeing that accurate event counts can be obtained efficiently at every point in the execution.

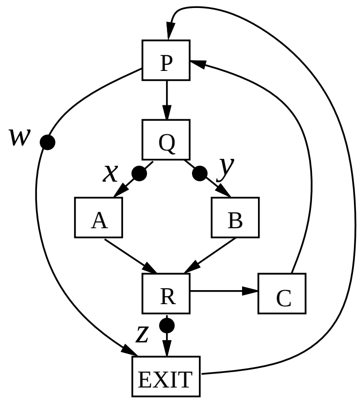
Event counting has a simple formalization. Associated with each basic block B is some constant $Events(B)$ that denotes the number of events in basic block B. This may be the number of instructions, cycles, stores/loads, etc. For any execution path, the goal of event counting is to keep track of the number of events in basic blocks that have executed fully. Because there are applications that require completely accurate event counts (such as debuggers and simulators), sampling the program counter to determine event counts will not suffice. Instead, we use instrumentation to provide accurate measurements. There are two basic instrumentation methods for event counting:

(1) Instrument the program with code to record the number of times each basic block executes (*i.e.*, maintain a basic block profile). Given basic block counts, the event count can be derived from the basic block count by summing the basic block counters, weighted appropriately (*e.g.*, if basic block B executes i times then add $i*Events(B)$ to the event count). Because of the potentially large number of counters that must be summed each time a query is made, this approach is suitable only if there are few queries of the event count during the execution of the program, or the query is made off-line.

(2) Instrument the program to record the number of events directly in an event counter. A straightforward approach is to prepend code to each basic block B that increments the event counter by $Events(B)$ each time that block B executes. This approach can incur high overhead (in the range of 200-300%) for programs with small basic blocks [4]. Furthermore, if instrumentation code is dynamically added to and deleted from programs by patching a basic block with a jump to a code stub rather than by rewriting the original code, the overhead can increase substantially [39].

This chapter defines a new approach to event counting that is similar to (2) above; however, our approach involves instrumenting control-flow edges (rather than basic blocks) in a procedure's control-flow graph. As shown in Chapter 3, instrumentation of edges in the control-flow graph can be used to profile programs with low execution-time overhead. Instrumentation code is placed along edges of the control-flow graph rather than in basic blocks because this gives greater opportunity to place the code in areas of lower execution frequency.

In profiling, each instrumented edge has its own counter, which is incremented whenever the edge is traversed in an execution. In event counting, the event counter is incremented by more than one instrumented edge, with different increments at each instrumented edge. The challenge is to determine a necessary and sufficient set of edges at which to place instrumentation code and to compute the increment associated with each instrumented edge. Figure 4.1 gives an example of efficient event counting. In this example, the number of events in a vertex (basic block) is



Edge e	*Increment(e)*
w	(+P +EXIT)
x	(+P +Q +A +R +C)
y	(+P +Q +B +R +C)
z	(−C +EXIT)

Vertex v	*QueryInc(v)*
P	(+P)
Q	(+P +Q)
A	(−R −C)
B	(−R −C)
R	(−C)
C	0
EXIT	0

Figure 4.1. An example of efficient event counting.

denoted by the vertex's label. The example control-flow graph has four instrumented edges (w, x, y, z), marked with black dots. For each instrumented edge e, there is an constant integer increment $Increment(e)$. This amount is added to the event counter each time edge e is traversed in an execution. It is not hard to verify that for any execution path from P to $EXIT$ the event count (as measured by the instrumented edges that appear in the execution) is correct. For example, the count for the execution path $P \rightarrow Q \rightarrow A \rightarrow R \rightarrow EXIT$ is $Increment(x) + Increment(z) = (+P+Q+A+R+C) + (-C+EXIT) = (+P+Q+A+R+EXIT)$.

An effect of our algorithm for counting events is that the value of the event counter is not guaranteed to be accurate at all points in the control-flow graph. In the example of Figure 4.1, if execution has only progressed to vertex Q then the event counter has not been incremented at all! On the other hand, if the path $P \rightarrow Q \rightarrow A$ has been executed then the event count overestimates the true count since the counts for vertices R and C have been taken into account. We solve this problem by computing for every vertex v a "query increment" that, when added to the event counter, produces the correct count at the end of the basic block corresponding to that vertex. That is, when a query is made just after executing vertex v, the value "$EventCounter + QueryInc(v)$" is the correct response. Figure 4.1 also shows the query increment for each vertex in the example control-flow graph.

The uninstrumented edges in the control-flow graph of Figure 4.1 form an (undirected) spanning tree of the graph. Our first result shows that for any spanning tree of the control-flow graph, instrumentation of non-tree edges is sufficient for event counting. We also give a simple algorithm that computes the increment value for each non-tree edge. It is possible to allow certain types of cycles in the set of uninstrumented edges, and we precisely characterize when a set of instrumented edges is necessary for event counting. Furthermore, we show how to accommodate control-edge events in addition to vertex (basic block) events.

This chapter is organized as follows. Section 4.1 presents some additional material on spanning tree theory needed in this chapter. Section 4.2 shows how to count events within a procedure efficiently. Section 4.3 shows how the results of Section 4.2 can easily be extended to count events interprocedurally. Section 4.4 discusses related work.

4.1. ADDITIONAL BACKGROUND

Let G be a CFG and let T be a spanning tree of G. Knuth [40] has proved a theorem about directed cycles that we will make use of here. Briefly stated, this theorem shows that the number of times any edge appears in a directed cycle in G is uniquely determined by the number of times each chord of the spanning tree T appears in the cycle.

Let D be a directed (not necessarily simple) cycle and let f be any edge in G. Let $Num(D,f)$ denote the number of times edge f appears in cycle D. Let $Same(D,f)$ and $Opp(D,f)$ denote the number of chords e in D such that f is in $C(e)$ and edges f and e are directed in the same direction or in opposite directions in the fundamental cycle $C(e)$, respectively. Keep in mind that these last two definitions are relative to the CFG G and spanning tree T of G. To simplify notation, explicit mention of T and G is omitted.

THEOREM (1). Let G be a CFG, let T be a spanning tree of G, and let D be a directed (not necessarily simple) cycle in G. For all edges f in CFG G, $Num(D,f) = Same(D,f) - Opp(D,f)$.

We give an example of an application of this theorem. Figure 4.2 shows the fundamental cycles associated with the spanning tree of the CFG from Figure 4.1. Each edge with a black dot is a chord of the spanning tree. Figure 4.3 shows a simple directed cycle EX. There are two chords in this directed cycle, edges x and z. Figure 4.3 also shows the fundamental cycles $C(x)$ and $C(z)$. Consider an edge $f \in EX$, so $Num(EX, f) = 1$. As Figure 4.3 shows, edge f either appears in $C(x)$ or $C(z)$ (but not both fundamental cycles) in the same direction as chord x or z. Therefore, $Same(EX, f) = 1$ and $Opp(EX, f) = 0$, so $Same(EX, f) - Opp(EX, f) = Num(EX, f)$. Consider an edge $f \notin EX$. It is clear that $Num(EX, f) = 0$. If f does not appear in either $C(x)$ or $C(z)$ then it follows directly that $Same(EX, f) = Opp(EX, f) = 0$. In this example, for each edge $f \notin EX$ that is in $C(x)$ or $C(z)$ (*i.e.*, edges $R \rightarrow C$ and $C \rightarrow P$), f is in the same direction as edge x in fundamental cycle $C(x)$ and is in the opposite direction from edge z in fundamental cycle $C(z)$. Therefore, for all edges f not in EX, $Same(D, f) - Opp(D, f) = 0$.

The number of times a vertex v appears in a directed cycle EX is equivalent to the number of edges e in EX such that $src(e) = v$. We denote this quantity by $Num(EX, v)$. An execution of a

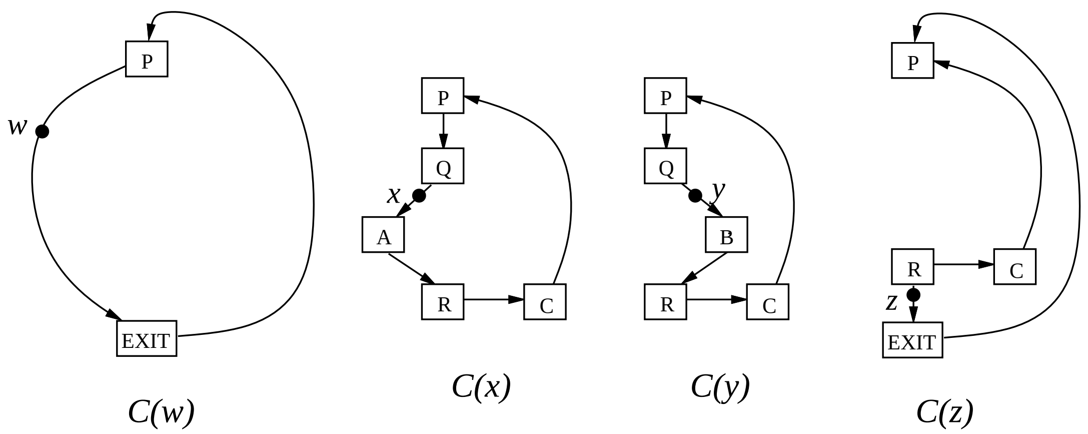


Figure 4.2. The fundamental cycles of the CFG from Figure 4.1with respect to the spanning tree in that figure.

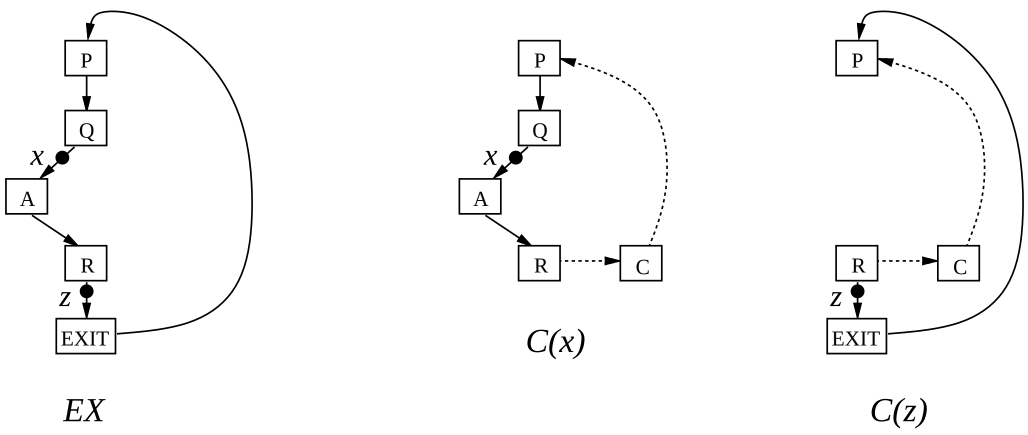


Figure 4.3. The directed cycle *EX* (from the CFG in Figure 4.1) and the two fundamental cycles associated with it.

procedure induces a directed path P in the CFG that starts at the root vertex (entry point) and ends at the *EXIT* vertex (return point). The addition of edge $EXIT \rightarrow root$ to the end of path P turns it into a directed cycle P', so an execution path is modelled by a directed cycle. It is clear that for any vertex v, the number of times vertex v appears in path P is equal to $Num(P', v)$.

4.2. INTRAPROCEDURAL EVENT COUNTING

We assume that each vertex v in the CFG contains a constant number of events, denoted by $Events(v)$. The goal of intraprocedural event counting is to find a (small) set of edges $E' \subseteq E$ and for each edge $e \in E'$ a constant $Increment(e)$ such that for any directed cycle EX in the CFG:

$$\sum_{e \in EX} Events(src(e)) = \sum_{\substack{e \in E' \\ \text{and } e \in EX}} Increment(e).$$

This definition does not address the problem of where accurate queries of the event count can be made in the execution path. The *EXIT* vertex is always an accurate query point since procedure execution ends at *EXIT*. As mentioned previously, not every vertex in the CFG is guaranteed to be an accurate query point. Given a set of edges E' and their increment values, we would like to find for each vertex w in the CFG, a constant $QueryInc(w)$ such that for any *incomplete* execution path *IX* starting at the *root* vertex and ending at vertex *w*:

$$\Big(\sum_{e \in IX} Events(src(e))\Big) + Events(w) \quad = \quad \Big(\sum_{\substack{e \in E' \\ \text{and } e \in IX}} Increment(e)\Big) + QueryInc(w)$$

Section 4.2.1 shows that placing instrumentation code on the chords of any spanning tree of the CFG is sufficient for event counting.[1] Section 4.2.1 also shows show to determine the increment for a chord *e* straightforwardly from its fundamental cycle $C(e)$. Section 4.2.2 describes how to accommodate edge events in this framework by the operation of *edge-splitting*. Section 4.2.3 presents a depth-first search algorithm that computes the increment for all chords in linear time. Section 4.2.4 supplies an exact structural characterization of when a set of edges is necessary and sufficient for event counting and shows how the algorithm in Section 4.2.3 can be easily adapted to compute increments in the exact case. Finally, Section 4.2.5 shows how to compute the query increment for any vertex in the CFG.

4.2.1. Computing Edge Event Increments

The increment value for a chord *e* is defined by its fundamental cycle $C(e)$ and the direction of the edges in this cycle. Given a chord *e*, we define two sets:

$PipeSame(e) =$

$\{\, v \mid$ there is a pipe at v in $C(e)$ whose edges are in the same direction as e in $C(e)\,\}$

$PipeOpp(e) =$

$\{\, v \mid$ there is a pipe at v in $C(e)$ whose edges are in the opposite direction from $e\,\}$.

The event increment for a chord *e* is defined to be:

$$Increment(e) \quad = \quad \sum_{v \in PipeSame(e)} Events(v) \quad - \quad \sum_{v \in PipeOpp(e)} Events(v).$$

---

[1]As discussed in Chapter 3, spanning trees must be chosen carefully in order to reduce the execution overhead incurred by instrumenting the chords.

Of course, if $Increment(e) = 0$ then edge e does not have to be instrumented.

Figure 4.2 shows the fundamental cycle of each chord in the CFG from Figure 4.1 (with respect to the spanning tree in that figure). The increment for chord w is defined to be $(+P\ +EXIT)$ since there are pipes at both P and $EXIT$ in $C(w)$ and these pipes are in the same direction as edge w. The fundamental cycle $C(z)$ defines $Increment(z) = (-C\ +EXIT)$ since there is a pipe at vertex C in the opposite direction to edge z and a pipe at $EXIT$ in the same direction as z.

We now show that this method is correct. Let G be a CFG and let T be a spanning tree of G. Given any directed cycle EX in CFG G, we want to show that:

$$(*) \qquad \sum_{e \in EX} Events(src(e)) \quad = \quad \sum_{\substack{e \in E-T \\ \text{and } e \in EX}} Increment(e).$$

Given a vertex v, let $VPipeSame(EX,v)$ be the number of edges e in EX such that e is a chord and $v \in PipeSame(e)$, and let $VPipeSame(EX,v)$ be the number of edges e in EX such that e is a chord and $v \in PipeOpp(e)$. We show that

(●) for all vertices v, $VPipeSame(EX,v) - VPipeOpp(EX,v) = Num(EX,v)$.

This results implies (*) directly, since for every vertex v in the directed cycle EX, $Events(v)$ makes a positive contribution to

$$\sum_{\substack{e \in E-T \\ \text{and } e \in EX}} Increment(e)$$

$Num(EX,v)$ more times than it does negatively because $VPipeSame(EX,v) - VPipeOpp(EX,v) = Num(EX,v) > 0$. Furthermore, for every vertex v that is not in EX (*i.e.* $Num(EX,v) = 0$), $Events(v)$ makes an equal number of positive and negative contributions (because $VPipeSame(EX,v) - VPipeOpp(EX,v) = 0$). Figure 4.3 illustrates this point. The directed cycle EX contains two chords, edges x and z. Every vertex that appears in EX appears in a pipe in either $C(x)$ or $C(z)$ in the same direction as the cycle's chord. Vertex C, which is not in EX, appears in a pipe in both $C(x)$ and $C(z)$. In $C(x)$ the pipe is in the same direction as edge x but in $C(z)$ the pipe is in the opposite direction from z, so $Events(C)$ makes a positive contribution to $Increment(x)$ and a negative contribution to $Increment(z)$, cancelling each other.

The proof of (●) relies on Theorem (1), as stated in Section 4.1. Given this theorem it is easy to show (●) as we do now. We require the following definitions:

$$SrcSame(EX,v) = \sum_{\forall e\,:\,src(e)=v} Same(EX,e)$$

$$SrcOpp(EX,v) = \sum_{\forall e\,:\,src(e)=v} Opp(EX,e)$$

Consider any vertex v in G. By theorem (1), it follows that

$$SrcSame(EX,v) - SrcOpp(EX,v) = \sum_{\forall e\,:\,src(e)=v} Num(EX,e).$$

By definition, the number of times a vertex v appears in directed cycle EX is equal to the number of edges in EX with source v. That is,

$$\sum_{\forall e\,:\,src(e)=v} Num(EX,e) = Num(EX,v).$$

It follows from the above two points that $SrcSame(EX,v) - SrcOpp(EX,v) = Num(EX,v)$. Any fundamental cycle $C(e)$ that contains v must have exactly two edges incident on v. These edges form either a pipe, a fork or a join at v. A fork at v contributes one unit to both $SrcSame(EX,v)$ and $SrcOpp(EX,v)$. Let $Fork(EX,v)$ be the number of edges e in EX such that e is a chord and there is a fork at v in $C(e)$. A join at v clearly cannot contribute to either $SrcSame(EX,v)$ or $SrcOpp(EX,v)$. A pipe at v either contributes one unit to $SrcSame(EX,v)$, or one unit to $SrcOpp(EX,v)$, depending on its direction in $C(e)$. Therefore, the following relationships hold:

$$SrcSame(EX,v) = Fork(EX,v) + VPipeSame(EX,v)$$

$$SrcOpp(EX,v) = Fork(EX,v) + VPipeOpp(EX,v)$$

It follows that $SrcSame(EX,v) - SrcOpp(EX,v) = VPipeSame(EX,v) - VPipeOpp(EX,v)$. Therefore, $VPipeSame(EX,v) - VPipeOpp(EX,v) = Num(EX,v)$. □

4.2.2. Accommodating Edge Events

In addition to vertex events, it is also possible to have edge events. Given an edge e, let $Events(e)$ denote the number of events associated with the execution of edge e. Edge events can be easily accommodated in our framework by a transformation that is applied to the CFG before the spanning tree is found and the chord increments are computed. This transformation turns an edge event into a vertex event.

If an edge $e = v \rightarrow w$ has an event value $Events(e)$, then eliminate the edge $v \rightarrow w$, add a new vertex x_e, and add edges $v \rightarrow x_e$ and $x_e \rightarrow w$ to the CFG. The event value for vertex x_e is defined to be $Events(x_e) = Events(e)$. Any cycle that used to include the edge $v \rightarrow w$ will now include the pipe $v \rightarrow x_e \rightarrow w$. It is easy to verify that the event value for edge e will be correctly taken into account by this transformation.

4.2.3. Computing Edge Increments via a Depth-first Search

A naive algorithm for determining $Increment(e)$ for each chord e would simply traverse each fundamental cycle $C(e)$. Since each fundamental cycle is, in the worst case, of size $O(E)$ and there are $E - (V - 1)$ event edges, this naive algorithm would run in $O(E \times (E - V))$ worst-case time. Figure 4.4 presents an algorithm that uses a depth-first search of the spanning tree to determine the increments for all chords in time $O(E)$. There are two parameters to the depth-first search functions dfs_from_src and dfs_from_tgt: an integer *events* which is the current event increment and an edge e which is the current edge in the depth-first search. Function dfs_from_src visits all edges incident on $src(e)$, while dfs_from_tgt visits all edges incident on $tgt(e)$. Each chord is visited twice by the algorithm, once by dfs_from_src and once by dfs_from_tgt. Each tree edge is visited exactly once, except for the initial edge d, which is visited twice. The running time of this algorithm is clearly $O(E)$.

We now explain the algorithm and show that it correctly computes $Increment(e)$ for each chord e. At some point the depth-first search will have just traversed an acyclic path P that begins with edge d and ends with chord e. For each pipe (at vertex v) in P, $Events(v)$ has been added to

```
for each e ∈ E − T do Increment(e) := 0 od
let d be an edge in T in
   dfs_from_tgt( 0, d )
   dfs_from_src( 0, d )
ni

function dfs_from_tgt(integer events, edge e)
   let v = tgt(e) in
      if e ∈ T then
         for each f : f ≠ e and v = tgt(f) do   // join
            dfs_from_src( −events, f )
         od
         for each f : v = src(f) do   // pipe
            dfs_from_tgt( events + Events(v), f )
         od
      else   // e ∈ E − T
         Increment(e) := Increment(e) + events
      fi
   ni

function dfs_from_src(integer events, edge e)
   let v = src(e) in
      if e ∈ T then
         for each f : f ≠ e and v = src(f) do   // fork
            dfs_from_tgt( −events, f )
         od
         for each f : v = tgt(f) do   // pipe
            dfs_from_src( events + Events(v), f )
         od
      else   // e ∈ E − T
         Increment(e) := Increment(e) + events
      fi
   ni
```

Figure 4.4. Algorithm for determining *Increment(e)* for each chord *e* using a depth-first search of spanning tree *T* of graph *G*.

events. For each join or fork in *P*, *events* has been negated. As shown in lemma (1) (see the end of this section), if two edges in an acyclic path *P* are in the same direction then there are an even total number of forks and joins in the subpath of *P* delimited by (and including) these edges. Therefore, for each pipe (at vertex *v*) in *P* in the same direction as *e*, $Events(v)$ will be negated an even number of times (through the negation of *events*) making a positive contribution to $Increment(e)$ (when the assignment "$Increment(e) := Increment(e) + events$" takes place). Lemma (1) also shows that if the edges in *P* are in opposite directions then there are an odd total number of forks and joins in the subpath. Therefore, for any pipe (at vertex *v*) in *P* in the opposite direction from *e*, $Events(v)$ will be negated an odd numbers of times, making a negative contribution to $Increment(e)$.

To show that $Increment(e)$ is correctly computed for any chord *e*, there are two cases we must consider: $d \in C(e)$ and $d \notin C(e)$. In the former case, the depth-first search will reach *e* two times from edge *d* (by paths P_1 and P_2). The only edges shared by the two paths are *d* and *e*, and the union of the edges in P_1 and P_2 is $C(e)$. As a result, each pipe in $C(e)$ is traversed by the depth-first search. As shown above, pipes in the same direction as e make a positive contribution to $Increment(e)$ while pipes in the opposite direction from e make a negative contribution to $Increment(e)$, as desired.

Suppose that $d \notin C(e)$. The depth-first search will still reach e two times from edge d (by paths P_1 and P_2). However, in this case P_1 and P_2 share a common prefix Q containing edges that are not members of $C(e)$. Let f be the last edge in Q, let P'_1 be the suffix of P_1 (*i.e.*, $P_1 = Q \,||\, P'_1$) and let P'_2 be the suffix of P_2. Furthermore, let g_1 be the first edge of P'_1 and let g_2 be the first edge of P'_2. The only edge shared by P'_1 and P'_2 is e and the union of the edges in P'_1 and P'_2 is $C(e)$. It is clear that the event values for pipes present in P'_1 and P'_2 will be correctly accounted for in $Increment(e)$. We must show that the event values accumulated in prefix Q cancel out and that $Increment(e)$ is correctly computed when g_1 and g_2 form a pipe in $C(e)$.

Suppose that edge f is encountered by the call dfs_from_tgt(x, f) (the proof for dfs_from_src(x, f) is symmetrical). Figure 4.5 shows the three cases that can arise: g_1 and g_2 form a pipe, fork, or join at v. For each case, Figure 4.5 illustrates the value for *events* that will be associated with the function call for each edge.

(a) Edges g_1 and g_2 form a pipe at v. In this case, f must form a pipe with one of the edges and a join with the other edge. Without loss of generality, assume that f forms a join with g_1 and a pipe with g_2. The value of *events* is $(-x)$ for g_1 and is $(x + Events(v))$ for g_2. Because g_1 and g_2 form a pipe in $C(e)$, either both expressions will be added to $Increment(e)$ or subtracted from $Increment(e)$. Therefore, x will cancel out and $Events(v)$ will be correctly accounted for.

(b) Edges g_1 and g_2 form a fork at v. In this case, the value of *events* associated with both edges is $(x + Events(v))$. One of the edges must be in the same direction as e and the other in the opposite direction, so $(x + Events(v))$ will be both added to and subtracted from $Increment(e)$.

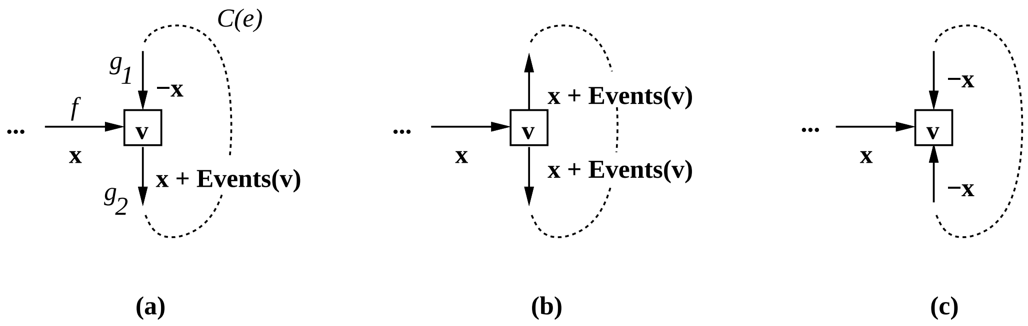


Figure 4.5. Entering a fundamental cycle via depth-first search.

(c) Edges g_1 and g_2 form a join at v. In this case, the value of *events* associated with both edges is $(-x)$. As in case (b), $(-x)$ will be both added to and subtracted from $Increment(e)$.

□

LEMMA (1). Let e and f be the first and last edges in an acyclic path P ($e \neq f$). If e and f are in the same direction in P then the number of forks in P is equal to the number of joins in P. If e and f point towards each other in P then the number of joins in P is one greater than the number of forks in P. If e and f point away from each other in P then the number of forks in P is one greater than the number of joins in P.

PROOF. The proof for the three claims is easily accomplished by induction on the length of P.

Base case: $|P| = 2$. It is clear that e and f form either a pipe, fork, or join, depending on the directions.

Induction Step: Suppose that all three claims hold for paths of length less than n ($n > 2$). Let P be a path of length $n+1$, let g be the n^{th} edge in P, and let P' be the path containing the first n edges of P. We prove the three claims by case analysis on the direction of edges e and g:

(1) Edges e and g are in the same direction. By the Induction Hypothesis, P' contains the same number of forks as joins. There are three possible cases: (a) g and f form a pipe, so the number of joins and forks in P is the same as in P'. In this case, edges e and f point in the

same direction; (b) g and f form a join, so P contains one more join than fork. In this case, edges e and f point towards each other; (c) g and f form a fork, so P contains one more fork than join. In this case, edges e and f point away from other.

(2) Edges e and g point towards each other By the Induction Hypothesis, P' contains one more join than fork. There are two possible cases: (a) g and f form a pipe, so P contains one more join than fork. In this case, edges e and f point towards each other; (b) g and f form a fork, so P contains the same number of forks and joins. In this case, edges e and f point in the same direction.

(3) Edges e and g point away from each other. The proof for this case is symmetric to the proof for case (2).

□

4.2.4. Exact Characterization

Section 4.2.1 showed that if $E-E'$ is a spanning tree of the CFG then E' is sufficient for event counting. A spanning tree is a maximal subset of edges without a (undirected) cycle. This section shows that it is possible to allow certain types of cycles in $E-E'$.

If a fundamental cycle $C(e)$ contains no pipe (is pipeless) then $Increment(e) = 0$, which implies that edge e does not have to be instrumented at all. This leads us to the following observation: if every simple cycle in $E-E'$ is pipeless then E' is sufficient for event counting. Suppose that $E-E'$ is a spanning subgraph of the CFG and that every simple cycle in $E-E'$ is pipeless (note that any subgraph may be turned into a spanning subgraph without introducing any new cycles into the subgraph). Let $F \subseteq E-E'$ be a set of edges such that $E-(E' \cup F)$ is a spanning tree of the CFG. By the results of Section 4.2.1, $E' \cup F$ is sufficient for event counting. However, for every edge $f \in F$, the fundamental cycle $C(f)$ must be pipeless, so $Increment(f) = 0$. Therefore, E' is sufficient for eventing counting.

As discussed in Chapter 3, the problem of finding a minimal size E' such that every simple cycle in $E-E'$ is pipeless appears to be a difficult problem to solve efficiently. This problem shares some characteristics with other NP-complete cycle-breaking problems such as Feedback

Arc Set and Uniconnected Subgraph [23].

Not surprisingly, the condition that every simple cycle in $E - E'$ is pipeless also is necessary for event counting. If $E - E'$ contains a simple cycle with a pipe then it is possible to find two executions EX_1 and EX_2 that exhibit different numbers of events but for which the event count computed by the instrumented points is the same. We refer the reader back to Figure 3.8 of Chapter 3.

4.2.5. Computing Query Increments

As mentioned before, there is a trade-off between efficiently counting program events and the number of points in a program at which queries are guaranteed to give correct results. If every basic block updates the event counter then queries of the event counter are accurate at any basic block. On the other hand, updating the event count only at the chord edges of the spanning tree leads to fewer points at which queries are accurate. This section shows that this problem can be easily solved by computing a *query increment*, $QueryInc(w)$, for each vertex w in the CFG. If a query is made just after executing vertex w then $QueryInc(w)$ is simply added to the event counter to obtain the correct event count (the event counter is *not* updated by this operation). That is, for any *incomplete* execution path IX starting at the *root* vertex and ending at vertex w:

$$\left(\sum_{e \in IX} Events(src(e)) \right) + Events(w) \quad = \quad \left(\sum_{\substack{e \in E' \\ \text{and } e \in IX}} Increment(e) \right) + QueryInc(w)$$

Computing $QueryInc(w)$ is an easy matter. Let T be the set of spanning tree edges. We simply add an edge $e_w = w \rightarrow root$ to the CFG and treat it as a member of $E - T$. This edge e_w is a chord of the spanning tree T with fundamental cycle $C(e_w)$. $QueryInc(w)$ is then defined to be $Increment(e_w)$.[2] The addition of edge e_w to any incomplete path IX from the *root* vertex to w yields a directed cycle IX' that contains the same set of vertices as IX. Therefore, by the results

---

[2]If the query is to be made at the beginning of the basic block associated with vertex w then $QueryInc(w)$ should be decremented by $Events(w)$. If the query is to be made at some intermediate point in the block then $QueryInc(w)$ can be adjusted by the appropriate amount.

of Section 4.2.1 it follows that for any incomplete path IX ending with vertex w:

$$\Big(\sum_{e \in IX} Events(src(e))\Big) + Events(w) \;=\; \sum_{\substack{e \in E-T \\ \text{and } e \in IX'}} Increment(e) \;=$$

$$\Big(\sum_{\substack{e \in E-T \\ \text{and } e \in IX}} Increment(e)\Big) + Increment(e_w)$$

Figure 4.6 illustrates this process on the example CFG from Figure 4.1. To compute $QueryInc(B)$ the CFG is augmented with a chord edge $b = B \rightarrow P$, defining a fundamental cycle $C(b)$. $QueryInc(B) = Increment(b) = (-R - C)$. To compute the query increment for all vertices in the CFG, we first add chord edge $e_v = v \rightarrow root$ for each vertex v and then apply the algorithm of Figure 4.4 to determine the increments (for all chord edges).

Note that query increments can also be used to correctly update the event counter when execution of a procedure terminates early (*i.e.*, not at the *EXIT* vertex) due to an exceptional conditional or interprocedural transfer of control (such as `setjmp/longjmp`). If a procedure exits early at vertex v then simply update the event counter by adding $QueryInc(v)$ to it.

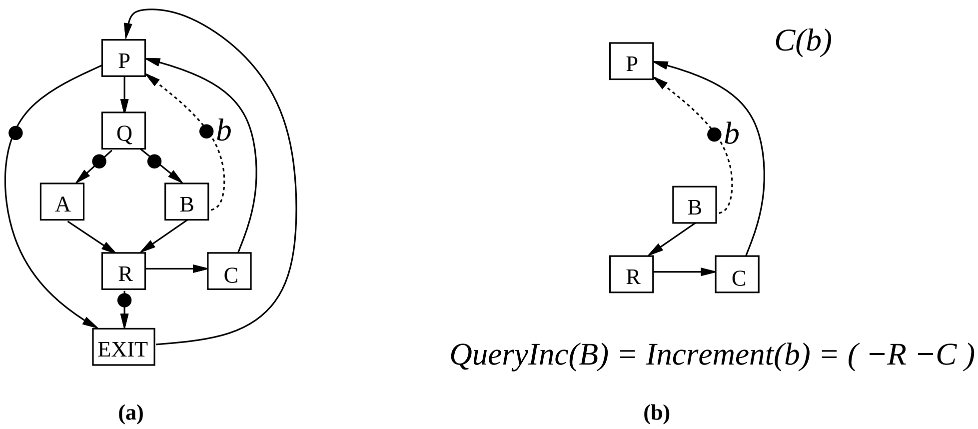


Figure 4.6. Computing *QueryInc(B)*.

4.3. INTERPROCEDURAL EVENT COUNTING

The previous section addressed how to count events within a procedure efficiently. This section describes two problems in efficiently counting events across procedure boundaries and their solutions.

The first problem is to maintain a global counter that counts the number of events executed throughout the entire program, with the capability to accurately query the counter at any place in the program. This problem is easily solved using the results of the previous section. The first step computes a spanning tree and determines the chord edge increments for each procedure's CFG. The second step ensures that the event count is correct whenever a procedure passes control to another procedure by treating procedure calls as queries: Immediately before the call the event counter is incremented by the query increment for the call, and immediately after the call the event counter is decremented by the same value (to ensure that counts will be correct after the call).[3] To query the event counter at vertex w in procedure X, we need only add $QueryInc(w)$ to the event counter.

Without the second step, queries of the event counter would be prohibitively expensive, as the following example illustrates. Suppose that vertex B in the CFG of Figure 4.1 contains a call to procedure X. When B passes control to X the event count is off because $Increment(y)$ has already accounted for vertices R and C, even though they have not executed. Therefore, to make an accurate query from vertex w in procedure X, the query must add both $QueryInc(w)$ and $QueryInc(R)$ to the event counter. In general, without the second step, the query may have to add in a query increment for each active procedure.

The second problem is to maintain a counter $cnt_{P,s}$ for each call site ($s : call\ Q$) in procedure P that tracks the number of events executed by the procedure Q (and procedures called transitively) when called from s in procedure P. A number of counters are used to solve this problem.

[3]For each call, it should be possible to eliminate either the increment or decrement by incorporating either one into the increment of nearby chords.

First, there is a global counter cnt_P for each procedure P. This counter is initialized to 0 at the beginning of program execution and counts the number of events executed by P and all procedures (transitively) called by P. However, cnt_P is only updated inside of procedure P. Second, for each (procedure P, call site $s : call\ Q$) pair, there is a global counter $cnt_{P,s}$ and a counter $Lcnt_{P,s}$, local to procedure P. The counter $cnt_{P,s}$ is initialized to 0 at the beginning of execution and $Lcnt_{P,s}$ is initialized as described below.

Counter increments are determined as follows. For each procedure P, a spanning tree is found and chord increments are computed. These increments are for events solely inside P and update the counter cnt_P. For each call site ($s : call\ Q$) in procedure P, the assignment "$Lcnt_{P,s} := cnt_Q$" is placed immediately before the call, and the assignments "$cnt_{P,s} := cnt_{P,s} + (cnt_Q - Lcnt_{P,s})$; $cnt_P := cnt_P + (cnt_Q - Lcnt_{P,s})$" are placed immediately after the call.

It is not difficult to see that this solution is correct by induction over the calling history of the program (as represented by a call tree). We show the base case. The induction step is quite similar. In the base case (at the leaves of the call tree) we have a call ($s : call\ Q$) from procedure P such that procedure Q makes no calls. By recording cnt_Q in $Lcnt_{s,P}$ before the call, we ensure that the difference ($cnt_Q - Lcnt_{P,s}$) is the number of events executed by procedure Q when called by P at s.

A query of a call site counter $cnt_{P,s}$ will reflect the number of events for all terminated calls made from that call site. The effect of still active calls is not reflected in the count until after the calls terminate. Within a procedure P, we can apply the query increment to cnt_P to get an accurate event count for cnt_P. However, outside of procedure P, the value of cnt_P will not be current (*i.e.*, reflect the number of events executed by P and procedures called by P) since cnt_P is only updated inside of procedure P.

4.4. RELATED WORK

As discussed in Chapter 3, there are a number of works on the related topic of efficiently profiling programs with instrumentation [4, 24, 40, 41, 55, 63, 64]. All of these (except [64]) use the spanning tree to determine a (small) set of points in a control-flow graph at which to place counters so that full and accurate vertex (basic block) profiles or edge profiles can be derived from the measured points. Knuth's theorem, stated in Section 4.1, shows that the number of times each control-flow edge appears in an execution is uniquely determined by the number of times each chord (of some spanning tree) appears in the execution. For edge profiling, this means that counters only have to be associated with chord edges, as described in Chapter 3. We have shown that this theorem can also be applied to event counting. Chapter 3 characterized when a set of edge counters is necessary and sufficient for vertex profiling [6]. This characterization is structurally equivalent to the characterization for event counting given in Section 4.2.4.

Mellor-Crummey and LeBlanc describe what they call a *software instruction counter* [49]. This term is misleading, since the software instruction counter does not actually count the number of instructions that have executed. Rather, the software instruction counter is a pair (*PC*,*SIC*) where *PC* is the program counter and *SIC* is a counter that is incremented for each backwards branch and procedure call executed. Because a program counter's value can only be reused if a backwards branch is taken or a chain of recursive calls is made, this pair of values uniquely identifies a particular instruction in a program's execution history. Such a counter has utility in cyclic debugging where one is interested in repeated executions and stopping at a particular state. However, as defined, the software instruction counter cannot be used to count the number of instructions executed.

Chapter 5

BACKGROUND FOR SLICING, DIFFERENCING, AND INTEGRATION

This chapter defines the goals of slicing, differencing, and integration. As the statements of these operations will make clear, each is an undecidable problem because each requires reasoning about when the behavior of one statement of a program affects the behavior of another statement. However, as with all undecidable problems of program behavior, it is possible to determine a safe approximation to the problem under consideration. As mentioned in Chapter 1, most previous work on slicing, differencing and integration has been limited to programs with structured control-flow. Chapters 6 and 7 describe conservative algorithms for slicing, differencing and integration in the presence of more complex control-flow.

To simplify our presentation and focus on the problems of slicing, differencing, and integration of programs with arbitrary control-flow, we consider a simplified language with the following characteristics: Expressions contain only scalar variables and constants; statements are either assignment statements, jump statements (*e.g.*, **break, halt, goto**), output statements, conditional statements (**if-then** or **if-then-else**), or loops (**while** and **repeat**). It is easy to generalize our techniques to handle languages with *N*-way branch constructs, such as **case** statements, and other looping constructs. The problems of slicing, differencing, and integration in the presence of multiple procedures [8, 32, 34, 36], non-scalar variables, and other language features [10] are orthogonal to the problems introduced by complex control-flow.

Section 5.1 reviews the control-flow graph representation that is used in the succeeding chapters and its execution semantics. Section 5.2 discusses the *dependence relationships* in the control-flow graph that form the foundation of our algorithms. Section 5.3 formally defines the goal of slicing. Section 5.4 describes the notion of corresponding components in different programs and Section 5.5 formally defines the goals of differencing and integration.

5.1. CONTROL-FLOW GRAPHS AND SEMANTICS

This section defines the control-flow graph, the standard translation from a program to a control-flow graph, and the execution semantics of the control-flow graph. The control-flow graph representation for slicing, differencing, and integration differs from the representation for profiling and tracing in two major respects: vertices in the former represent single instructions rather than basic blocks of instructions, and the root vertex of the former is a special *ENTRY* vertex, rather than the vertex that represents the first basic block to be executed. Paths and cycles in graphs are now assumed to be directed.

5.1.1. The Control-flow Graph

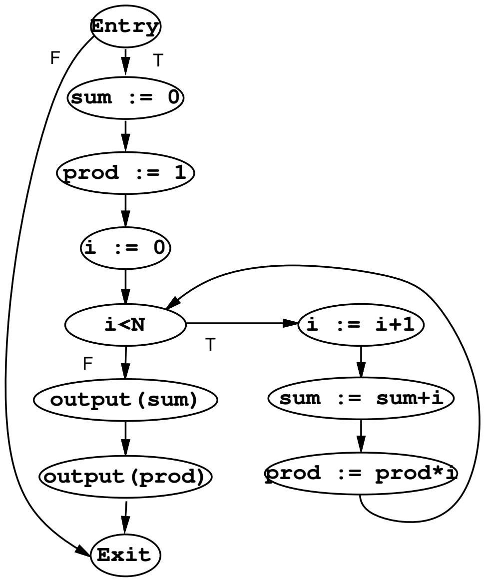
A *control-flow graph* (CFG) is any directed, rooted graph that satisfies the following conditions. The CFG has three types of vertices: Statement vertices (either assignment statements or output statements), which have one successor; predicate vertices, which have one *true*-successor and one *false*-successor; and an *EXIT* vertex, which has no successors. The root of the CFG is the *ENTRY* vertex, which is a predicate that has the *EXIT* vertex as its *false*-successor. Every vertex is reachable from the *ENTRY* vertex, and the *EXIT* vertex is reachable from every vertex. Edges in the CFG are labeled; the outgoing edges of a predicate vertex are labeled *true* or *false* (as appropriate) and the outgoing edge of a statement vertex is labeled *null*.

5.1.2. Standard Control-flow Translation

In the standard translation from a program to a CFG, the CFG includes a vertex for every assignment statement, output statement, and predicate in the program. The edges of the CFG represent the flow of control (the *ENTRY* vertex's *true*-successor is the first statement in the program). Jump statements (such as **break** and **goto**) are not represented directly as vertices in the CFG; instead, they are represented indirectly in that they affect the flow of control, and therefore the targets of some CFG edges. Figure 5.1 presents an example program and its CFG.

```
begin
  sum := 0
  prod := 1
  i := 0
  while i < N do
    i := i+1
    sum := sum+i
    prod := prod*i
  od
  output(sum)
  output(prod)
end
```

(a) Example program



(b) Example control-flow graph

Figure 5.1. A program and its control-flow graph.

Figure 5.2 presents an attribute grammar for the language under consideration, in which the attributes are used to define the translation from a program to its CFG. The grammar is given in the style used in [60], in which the underlying context free grammar defines a program's *abstract* (rather than concrete) syntax. Operator names are used to identify productions uniquely. Each production in the grammar is of the form "x_0: $op(x_1\ x_2\ \cdots\ x_k)$", where *op* is an operator name and each x_i is a nonterminal. Every nonterminal has a synthesized attribute *entry* and an inherited attribute *cont*, both of which represent vertices in the CFG. The constructor Pred(t,v,w) creates a predicate vertex with text t, *true*-successor v, and *false*-successor w, while the constructor FallThrough(t,v) creates a fall-through vertex with text t and successor v.

The notation "$attr = \{ \cdots \}$" is an upward remote attribute reference; its value is the first instance of a set element encountered on the path to the root in the abstract syntax tree. For example, a **break** statement passes control to the continuation of the innermost enclosing loop. A

```
prog:
   Program ( seq ) {
      prog.entry = Pred( "ENTRY", seq.entry, seq.cont)
      prog.cont = seq.cont = FallThrough( "EXIT", null)
   };

seq:
   NullSeq () {
      seq.entry = seq.cont
   }
|  Sequence ( stmt seq ) {
      seq1.entry = stmt.entry
      stmt.cont = seq2.entry
      seq2.cont = seq1.cont
   };

stmt:
   NullStmt () {
      stmt.entry = stmt.cont
   }
|  While ( expr seq ) {
      stmt.entry = Pred( "expr", seq.entry, stmt.cont)
      seq.cont = stmt.entry
   }
|  Repeat ( seq expr ) {
      stmt.entry = seq.entry
      seq.cont = Pred( "expr", stmt.cont, seq.entry)
   }
|  IfThen ( expr seq ) {
      stmt.entry =
          Pred( "expr", seq.entry, stmt.cont)
      seq.cont = stmt.cont
   }
|  IfThenElse ( expr seq seq ) {
      stmt.entry =
          Pred( "expr", seq1.entry, seq2.entry)
      seq1.cont = seq2.cont = stmt.cont
   }
|  Assign ( ID expr ) {
      stmt.entry =
          FallThrough( "ID := expr", stmt.cont)
   }
|  Label ( ID ) {
      stmt.entry = stmt.cont
      insert(ID, stmt.entry)
   }
|  Break () {
      stmt.entry = { Repeat.cont, While.cont }
   }
|  Halt () {
      stmt.entry = { Program.cont }
   }
|  Goto ( ID ) {
      stmt.entry = lookup(ID)
   };
```

Figure 5.2. Abstract syntax for the language under consideration with attribution that defines the control-flow graph. The notation “expr” means some appropriate representation of the expression; for example, its parse tree.

global symbol table (with operations *insert* and *lookup*) is used to manage the control-flow between **Goto** and **Label**. In a pure attribute grammar the symbol table would be threaded through the abstract syntax tree.

The grammar presented in Figure 5.2 makes precise the relationship between a program’s abstract syntax tree and its CFG. Every vertex in the CFG is associated with the *stmt* production that created the vertex. In the remainder of the thesis, we use the term *program component* to refer to an instance of a *stmt* production in a program (*i.e.*, a node in the abstract syntax tree) that creates a CFG vertex (some productions, such as **Label** do not create a vertex). Given a *stmt* subtree T, $vert(T)$ denotes the CFG vertex associated with T's root production.

We note that there are programs whose standard control-flow translation does not yield a CFG. For example, it is possible to create code that is not reachable from *ENTRY* or from which the *EXIT* vertex is unreachable, or to define a program whose control-flow translation is not well-defined (for example, a **goto** to an undefined label or a **break** that is not enclosed in a loop). We consider only programs that yield a CFG under the standard translation.

5.1.3. Control-flow Graph Semantics

The operational semantics for the CFG is defined as follows: Execution starts at the *ENTRY* vertex (which always evaluates to *true*), with an initial state σ; at any moment there is a single point of control together with a state mapping variables to values; the execution of each statement or predicate vertex passes control to a single successor. The execution of an assignment statement changes the state. Execution terminates normally if *EXIT* is reached (execution can fail to terminate normally if the program includes an infinite loop or an exception such as division by zero). An execution of CFG G on initial state σ is denoted by $G(\sigma)$.

For an execution $G(\sigma)$, we characterize the *behavior* at a vertex by the sequence of values that arise at that vertex[1]. This is defined as follows: For an assignment statement vertex, the sequence of values assigned to the left-hand-side variable; for an output statement, the sequence of values output; and for a predicate vertex, the sequence of boolean values to which the predicate's boolean expression evaluates. $G(\sigma)(v)$ denotes the sequence of values that arise at vertex v in execution $G(\sigma)$.

A common thread in the operations of slicing, differencing, and integration is the comparison of the behavior of components across programs. The following definition defines what we mean

---

[1]Note that our definition of a vertex's behavior differs from the more standard definition, which characterizes a vertex's behavior as the sequence of *states* that arise at that vertex during an execution (where a state associates a value with *every* variable in the program).

for two vertices in different CFGs to have equivalent behavior. This definition will be used to define the goals of slicing, differencing, and integration.

DEFINITION (equivalent behavior of vertices). Vertices v_G and v_H of CFGs G and H, respectively, have *equivalent behavior* iff all the following hold:

- For all σ such that both $G(\sigma)$ and $H(\sigma)$ terminate normally, $G(\sigma)(v_G) = H(\sigma)(v_H)$.
- For all σ such that neither $G(\sigma)$ nor $H(\sigma)$ terminates normally, $G(\sigma)(v_G)$ is a prefix of $H(\sigma)(v_H)$, or vice versa.
- For all σ such that $G(\sigma)$ terminates normally but $H(\sigma)$ does not, $H(\sigma)(v_H)$ is a prefix of $G(\sigma)(v_G)$.
- For all σ such that $H(\sigma)$ terminates normally but $G(\sigma)$ does not, $G(\sigma)(v_G)$ is a prefix of $H(\sigma)(v_H)$.

5.2. PROGRAM DEPENDENCES

Slicing, differencing, and integration make use of *program dependences*, relations between vertices in a CFG. We consider two types of dependences: control dependences and flow dependences.

DEFINITION (postdominance). Let v and w be vertices in a CFG G. Vertex w *postdominates* vertex v, denoted by w **pd** v, iff $w \neq v$ and w is on every path from v to the *EXIT* vertex. Vertex w *postdominates* the L-branch of predicate vertex v (where L is either *true* or *false*), denoted by w **pd** (v,L), iff w is the L-successor of v or w postdominates the L-successor of v. While no vertex can postdominate itself, a vertex can postdominate its own L-branch. The immediate postdominator of a vertex v, denoted by ipd(G,v), is the postdominator of v such that there is no vertex w such that ipd(G,v) **pd** w **pd** v.

DEFINITION (control dependence). Let v and w be vertices in a CFG. Vertex w is directly L-control dependent on v (written $v \longrightarrow_c^L w$) iff w postdominates the L-branch of v and w does not postdominate v. Intuitively, if $v \longrightarrow_c^L w$, then whenever v executes and evaluates to L, w will eventually execute, barring abnormal termination. Furthermore, if v executes and does not evaluate to L, w might not execute.

DEFINITION (transitive control dependence). Given a CFG G and vertex v, transCD(G,v) denotes the set of vertices that are reflexively and transitively control dependent on v (*i.e.*, $\{ w \mid v \longrightarrow_c^* w \}$). It is well known that this set can be equivalently defined as the set of vertices reachable from v in G via a path that does not include ipd(G,v).

DEFINITION (flow dependence). Let v and w be vertices in a CFG G. There is a *flow dependence* from vertex v to vertex w (written $v \longrightarrow_f w$) iff vertex v assigns to variable x, vertex w uses x, and there is a path in G from v to w that does not include an assignment to x (excluding v and w).

The *program dependence graph* (PDG) is a graph that contains the same vertex set as the CFG (except for the *EXIT* vertex) [17]. The edges of the PDG are the control and flow dependences, as defined above.[2] Given a CFG G, let PDG(G) denote G's program dependence graph. The control dependence subgraph of PDG(G), denoted by CDG(G), contains only control dependence edges.

Figure 5.3 shows the program dependence graph of the CFG in Figure 5.1(a). Solid edges are control dependences and dashed edges are flow dependences.

5.3. SLICING PROGRAMS

Program slicing, a program transformation originally defined by Mark Weiser [71], is useful in program debugging [42], program maintenance [30], and other applications that involve understanding program behavior [31]. Given a program component c and a set of variables V, the goal of slicing (as defined by Weiser) is to create a *projection* of the program (by eliminating some

[2]In addition to control and flow dependences, program dependence graphs usually include either def-order dependences [29] or output and anti-dependences [17]. These additional edges are not needed for slicing, and so are omitted from the definition given here. We also do not need to distinguish between loop-independent and loop-carried dependences (which are ill-defined for irreducible control-flow). We will introduce a new type of edge needed for differencing and integration in Chapter 7.

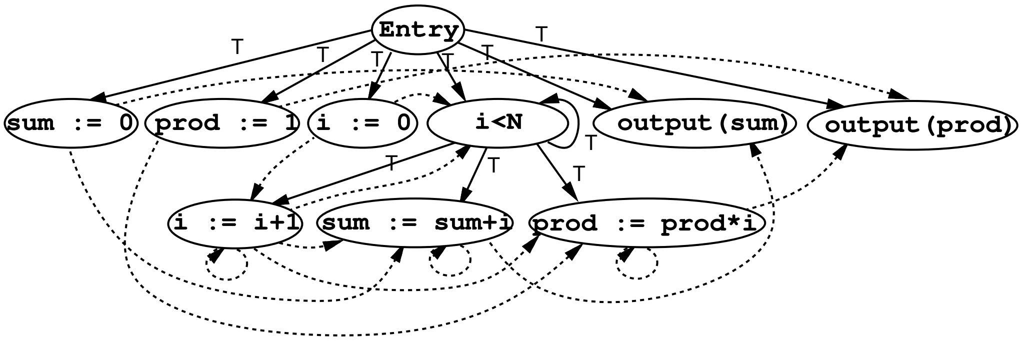


Figure 5.3. The program dependence graph of the CFG in Figure 5.1(a). Solid edges are control dependences and dashed edges are flow dependences.

statements), such that the projection and the original program compute the same values for all variables in *V* at component *c*. Implicit in the goal of slicing is that the program projection be as small as possible. The program shown in Figure 5.4(a) computes the sum and product of the numbers from 1 to *N*[3]. Figure 5.4(b) shows the result of slicing the example program with respect to the statement **output**(*prod*) and the variable *prod*. For any value of *N*, the example program and the program projection compute the same value for variable *prod* in the **output** statement.

We consider a slightly restricted version of slicing in which the set of variables *V* includes only variables used or defined at component *c*. However, following the example of Reps and Yang [62], we strengthen the goal of slicing by requiring that each component of the program projection have equivalent behavior to its corresponding component in the original program. That is, given a program *P* and component *c*, the goal of slicing is to find a program projection *Q* that includes *c* such that each component of *Q* has equivalent behavior to its corresponding

[3]In the example program, variable *N* is used without being explicitly initialized. It is assumed that such variables get their values from an initial state on which the program is executed.

(a) Example Program	(b) Result of slicing with respect to **output**(*prod*)

begin
  sum := 0
  prod := 1
  i := 0
  while *i* < *N* **do**
    i := *i*+1
    sum := *sum*+*i*
    prod := *prod***i*
  od
  output(*sum*)
  output(*prod*)
end
	begin
	
	  prod := 1
	  i := 0
	  while *i* < *N* **do**
	    i := *i*+1
	
	    prod := *prod***i*
	  od
	
	  output(*prod*)
	end

Figure 5.4. An example program, and the result of slicing with respect to **output**(*prod*).

component in P. The program projection in Figure 5.4(b) meets this stronger requirement.

5.4. CORRESPONDING COMPONENTS

The operation of slicing creates a program Q from program P by eliminating components of program P. Therefore, the correspondence between components of P and Q is straightforwardly determined. However, the operations of differencing and integration take more than one program as input: Differencing compares the behavior of two different programs, while integration attempts to merge two variants of an original program in a satisfactory manner. Differencing and integration require that *corresponding* program components in different programs be identified in some way. A correspondence between programs *New* and *Old* is a 1-1 partial function f from components (vertices) of *New* to components of *Old* such that for all vertices v of *New*, either (1) $f(v)$ is a component of *Old* with the same abstract syntax and associated text as v, or (2) $f(v)$ is undefined. There are several ways in which a correspondence may be established:

- A language-based editor may be used to tag program components and track them over editing changes in order to maintain the correspondence, as suggested by Horwitz, Prins, and Reps [30].

- The correspondence can be computed directly from the two programs. For example, Horwitz suggests a number of approaches for computing a correspondence based on program dependences [31], while Yang gives an algorithm that finds a maximal syntactic match between two programs by dynamic programming [72].

We will not concern ourselves here with how a correspondence is established, relying on one of the above techniques to provide the mapping between programs.

5.5. DIFFERENCING AND INTEGRATION

Given two programs, *Old* and *New*, and a correspondence between components of the programs, the difference of *New* with respect to *Old* is a set of components whose behavior in *New* is different than in *Old*. In particular, the difference of *New* with respect to *Old* includes all components of *New* that have no corresponding component in *Old*, as well as all components in *New* whose corresponding component in *Old* has inequivalent behavior. Because it is impossible to identify this set exactly, a safe algorithm will also sometimes include components of *New* whose corresponding component in *Old* has equivalent behavior. Figure 5.5 shows two programs, *Old*

Old

```
begin
  sum := 0
  prod := 1
  i := 0
  while i < N do
     i := i+1
     sum := sum+i
     prod := prod*i
  od
  output(sum)
  output(prod)
end
```

New

```
begin
  sum := N            ←  •
  prod := 1
  i := 0
  while i < N do
     i := i+1
     sum := sum+i     ←  •
     prod := prod*i
  od
  output(sum)         ←  •
  output(prod)
end
```

Figure 5.5. Computing the differences of program *New* with respect to program *Old* Program *New* is annonated to show components that may exhibit different behavior from the corresponding components of *Old*.

and *New*. The assignment statement "*sum := 0*" in *Old* has been changed to "*sum := N*" in *New* so that *New* computes the sum of the numbers from $N+1$ to $N+N$. Differencing shows those components of *New* whose behaviors may be different.

Program integration takes three programs as input: an original program *BASE*, and programs *A* and *B*, which are variants of *BASE*. Correspondence mappings between *A* and *BASE*, and *B* and *BASE* are also required. Informally stated, the goal of integration is to automatically determine the changes in each variant with respect to *BASE*, and incorporate these changes, as well as the portion of *BASE* that is preserved in the variants, into a merged program *M*. Stated more formally, the goal of integration is to find a program *M* such that for any initial state σ on which *A*, *B*, and *BASE* all terminate normally, all of the following hold:

(1) $M(\sigma)$ terminates normally;

(2) If there are corresponding components v_A and v_{BASE} such that $A(\sigma)(v_A) \neq BASE(\sigma)(v_{BASE})$, then there is a component v in M such that $M(\sigma)(v) = A(\sigma)(v_A)$;

(3) If there are corresponding components v_B and v_{BASE} such that $B(\sigma)(v_B) \neq BASE(\sigma)(v_{BASE})$, then there is a component v in M such that $M(\sigma)(v) = B(\sigma)(v_B)$;

(4) If there is a component v in *BASE* with corresponding components in *A* and *B* such that the sequence of values at v in *A*, *B*, and *BASE* is the same then there is a component w in *M* such that $M(\sigma)(w) = BASE(\sigma)(v)$.

An additional goal of integration is that the program *M* should resemble the input programs as much as possible.

Figure 5.6 presents an example of a successful integration. Program *BASE* sums the integers from 1 to *N*. Variant *A* changes the initial assignment to the variable *sum* to compute the sum of the numbers from *N* to $N+N$ (rather than from 1 to *N*). Variant *B* adds code to compute the product of the numbers from 1 to *N*. Program *M* is a merged program that successfully merges the

BASE	*A*	*B*	*M*

begin
  sum := 0
  i := 0
  while *i* < *N* **do**
    i := *i*+1
    sum := *sum*+*i*
  od
  output(*sum*)
end
	begin
	  sum := *N* ← •
	  i := 0
	  while *i* < *N* **do**
	    i := *i*+1
	    sum := *sum*+*i* ← •
	  od
	  output(*sum*) ← •
	end
		begin
		  sum := 0
		  prod := 1 ← •
		  i := 0
		  while *i* < *N* **do**
		    i := *i*+1
		    sum := *sum*+*i*
		    prod := *prod***i* ← •
		  od
		  output(*sum*)
		  output(*prod*) ← •
		end
			begin
			  sum := 1
			  prod := 1
			  i := 0
			  while *i* < *N* **do**
			    i := *i*+1
			    sum := *sum*+*i*
			    prod := *prod***i*
			  od
			  output(*sum*)
			  output(*prod*)
			end

Figure 5.6. Integration of programs *A* and *B* with respect to *BASE*. Programs *A* and *B* are variants of program *BASE*. Those components marked with bullets in the variants represent components whose behavior is not equivalent to the corresponding component in *BASE*. Program *M* is an acceptable result of integration.

changes introduced in the variants and preserves the common behavior of all three (the computations involving induction variable *i*). Of course, there are input programs for which integration will not succeed because the changes made in variants *A* and *B* conflict with one another. For example, in Figure 5.6, if variant *B* had changed the assignment "*sum := 0*" in *BASE* to "*sum := 1*", this would conflict with the change to the assignment made in variant *A*.

Chapter 6

SLICING PROGRAMS WITH ARBITRARY CONTROL-FLOW

This chapter addresses the problem of slicing programs with unstructured control-flow, *i.e.*, programs that include constructs such as **break**, **continue**, and **goto**. Previous algorithms for slicing programs with unstructured control-flow are ad-hoc, overly conservative in their approach (that is, the projections they produce are unnecessarily large), and lack any accompanying proofs of correctness. We give a program-slicing algorithm based on program dependence graphs that correctly handles such programs, and we prove that the program projections produced by our algorithm meet the semantic goal of program slicing: each component in the projection has equivalent behavior to its corresponding component in the original program (including the point of the slice). Our algorithm works for programs with completely arbitrary control-flow, including irreducible control-flow [1].

Algorithms for slicing programs with *structured* control-flow have been defined by Weiser [71] and by the Ottensteins [52]. Neither of these algorithms works correctly for programs with unstructured control-flow. Lyle developed an ad-hoc algorithm for slicing programs in the presence of arbitrary control-flow but his algorithm is overly conservative, as described by Gallagher [21]. Gallagher refined Lyle's algorithm to produce smaller slices but his algorithm contains some errors that cause it to produce semantically incorrect program projections [21].

We focus on the Ottensteins' algorithm and consider the problems that arise if one tries to apply this algorithm to programs with unstructured control-flow. The Ottensteins' algorithm makes use of the control-flow graph and program dependence graph representations. There are two steps to the algorithm for slicing with respect to program component *c*. In Step 1, control and flow dependence edges are traversed in the reverse direction in the program dependence graph from the vertex corresponding to component *c* (we refer to this step as *backwards-closure*), identifying a set of vertices *S* in the program dependence graph. Step 2 produces the program projection by eliminating from the original program all components that do not correspond to a vertex in *S*.

If one uses the standard control-flow graph for programs with unstructured control-flow (*i.e.*, a graph in which a jump gives rise to a single control-flow edge—see Figure 6.1(b)), the program projections computed by the Ottensteins' algorithm may fail to meet the semantic goal of program slicing; that is, the projections may compute different values than the original program at the point of the slice. The problem is that the algorithm does not correctly detect when unconditional jumps in the program (such as the **break** in Figure 6.1) are required in the program projection in order to meet the semantic goal of slicing.

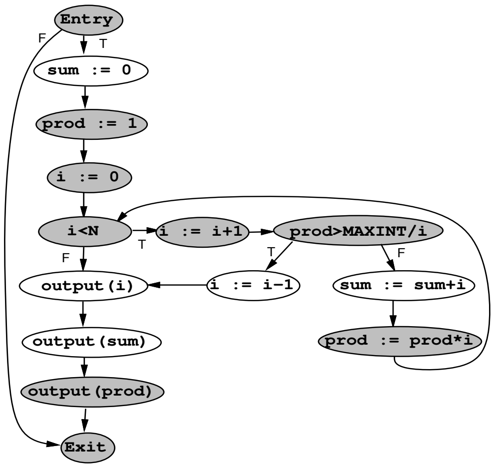
Example. Consider the program shown in Figure 6.1(a). Figure 6.1(b) shows the standard control-flow graph for this program, and Figure 6.1(c) shows the program dependence graph of this control-flow graph. In the two graphs, shading is used to indicate the vertices that would be identified by backwards-closure in the program dependence graph with respect to "**output**(*prod*)". Figure 6.1(d) shows the program projection obtained by eliminating all components not identified by the slicing algorithm. Because the **break** statement is absent from the projection, it does not satisfy the semantic goal (*i.e.*, for some values of *N* and *MAXINT*, different final values of *prod* will be output by the original program and by the projection). A projection that does satisfy the semantic goal (and that would be produced by the slicing algorithm defined in this chapter) is shown in Figure 6.1(e). □

Simply including a vertex for the **break** in the control-flow graph, such that the **break** vertex has a single successor, does not solve the problem. The **break** will still be omitted from the slice because in the program dependence graph the **break** vertex will have no outgoing flow or control dependence edges, and so there will be no path from the **break** to the vertex "**output**(*prod*)".

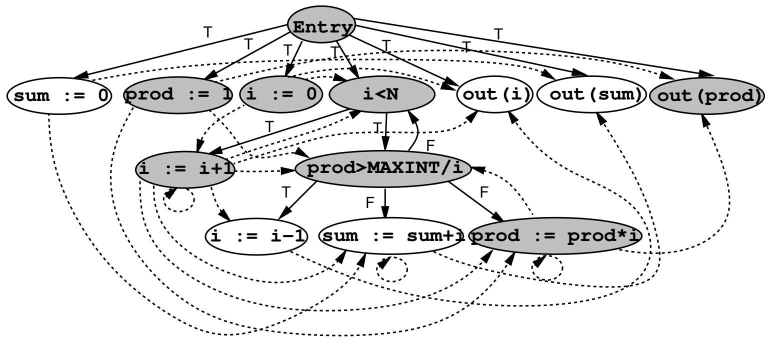
The main result of this chapter is a slicing algorithm for programs with unstructured control-flow, and a proof of the correctness of this algorithm; that is, we show that the program

```
begin
  sum := 0
  prod := 1
  i := 0
  while i < N do
    i := i+1
    if prod > MAXINT/i then
      i := i−1
      break
    fi
    sum := sum+i
    prod := prod*i
  od
  output(i)
  output(sum)
  output(prod)
end
```

(a) Example program



(b) Control-flow graph



(c) Program dependence graph

```
begin
  prod := 1
  i := 0
  while i < N do
    i := i+1
    if prod > MAXINT/i then
    fi
    prod := prod*i
  od
  output(prod)
end
```

(d) Incorrect projection

```
begin
  prod := 1
  i := 0
  while i < N do
    i := i+1
    if prod > MAXINT/i then
      break
    fi
    prod := prod*i
  od
  output(prod)
end
```

(e) Correct projection

Figure 6.1. An example program, its control-flow graph, its program dependence graph (solid arrows are control dependences, dashed arrows are flow dependences), the (incorrect) projection that would be computed using the Ottensteins' algorithm to slice with respect to **output**(*prod*), and the correct projection.

projections produced by the algorithm have the desired semantic property: Both the original program and the projection compute the same values at the point of the slice. The algorithm is in the style of the Ottensteins' algorithm in that it operates on a program dependence graph representation of a program; however, the program dependence graph is based on a control-flow graph in which a jump is represented as a pseudo-predicate vertex (that always evaluates to *true*). The jump vertex's *true*-successor is the target of the jump, and its *false*-successor is the vertex that represents the jump statement's continuation (that is, the vertex that would be the jump vertex's successor if it were a "no-op" rather than a jump). We are able to prove that by using this augmented control-flow graph, a projection of the program that has the desired semantic property can be formed.

This chapter is organized as follows. Section 6.1 presents our slicing algorithm and gives an outline of its proof of correctness. Section 6.2 fills in the details of the proof. Section 6.3 discusses the issues of *minimal* slices. Section 6.4 describes how to slice in the presence of looping and conditional control constructs other than those described in Chapter 5. Section 6.5 reviews related work in the area of slicing.

6.1. ALGORITHM AND PROOF SKETCH

In this section we present our slicing algorithm and sketch a proof that it produces program projections with the desired semantic property: Given program P and component c, our algorithm, $Slice(P, c)$, produces a projection Q including component c such that each component of Q has equivalent behavior to its corresponding component in P.

6.1.1. The Slicing Algorithm

Our slicing algorithm is similar to the Ottensteins' algorithm in that it uses backwards-closure in the program dependence graph (PDG) to identify the program components in the slice. In particular, given a PDG and a vertex v (corresponding to component c: $v = vert(c)$) from which to slice, Step 1 of our algorithm identifies the subset of the PDG's vertices from which there is a path along control and/or flow dependence edges to vertex v (*i.e.*, Step 1 computes the backwards reflexive transitive closure over control and flow dependences from vertex *v*). Step 2 of the algorithm creates a program projection by eliminating components from the original program *P* that do not correspond to the vertices identified in Step 1, resulting in program *Q*. For each vertex *w* that is not identified by Step 1 of the algorithm, Step 2 eliminates the *stmt* subtree *T* such that $vert(T) = w$. Eliminating a *stmt* subtree *T* that contains no **Label** subtrees is accomplished simply by replacing that subtree by the **NullStmt** subtree (Figure 6.2(a)). If *T* contains **Label** subtrees, they are sequenced together to replace *T* (Figure 6.2(b)). The order of the labels in the sequence is not important. If there is no **Goto** to a **Label** subtree in the program, then the **Label** subtree can be eliminated by replacing it with **NullStmt**.

The important difference between our algorithm and the Ottensteins' is that we use an *augmented* control-flow translation from the program to the CFG, from which the PDG is built. The translations for all the structured constructs (*i.e.*, **if-then, if-then-else, while**, and **repeat**) remain the same. However, jump statements are explicitly represented in the CFG as pseudo-predicate

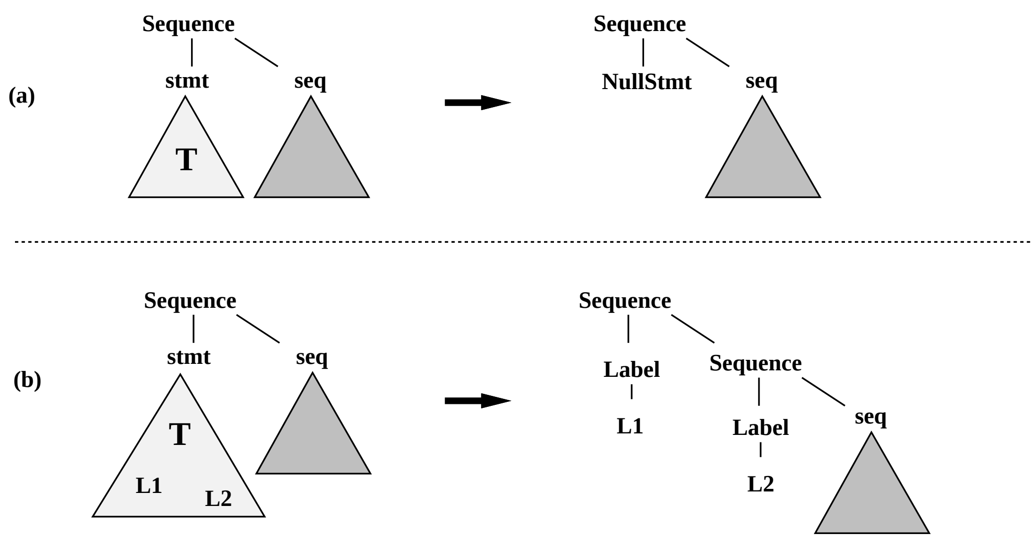


Figure 6.2. Eliminating a *stmt* subtree without labels (a) and with labels (b).

vertices that always evaluate to *true*. Figure 6.3 presents the translations of these constructs under the augmented translation. A jump vertex's *true*-successor is the target of the jump; its *false*-successor is the vertex that represents the jump statement's fall-through or continuation (that is, the vertex that would be the jump vertex's successor if it were a "no-op" rather than a jump). The outgoing false edge of a jump vertex is a "dummy" edge that is never actually traversed in an execution.[1] Representing a jump statement this way causes it to be the source of control dependence edges in the PDG. This in turn allows the jump vertex to be correctly included in the backwards-closure in the PDG.

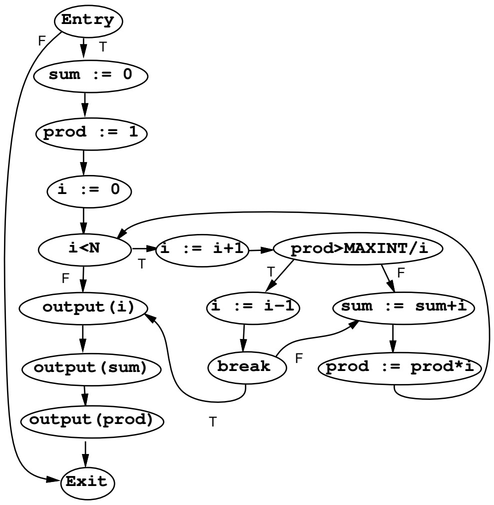
Example. Figure 6.4(a) repeats the program of Figure 6.1(a) and shows the program's augmented CFG. Figure 6.4(b) shows the vertices, control edges, and some of the flow edges of the corresponding PDG (flow edges that are not relevant to the backwards-closure with respect to "**output**(*prod*)" are omitted).

```
stmt:
   Break () {
      stmt.entry = Pred( "break", { Repeat.cont, While.cont }, stmt.cont)
   }
|  Halt () {
      stmt.entry = Pred( "halt", { Program.cont }, stmt.cont)
   }
|  Goto ( ID ) {
      stmt.entry = Pred( "goto ID", lookup(ID), stmt.cont)
   };
```

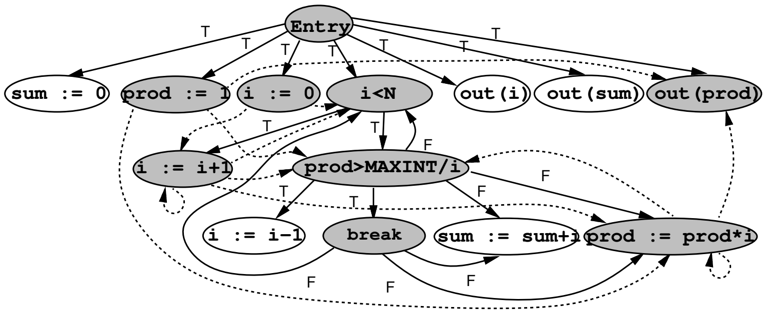
Figure 6.3. Augmented control-flow translations for jump statements.

[1] It is important to note that representing jump statements this way in the CFG does not change the semantics of the CFG as defined in Chapter 5. In particular, since a jump is treated as a predicate that always evaluates to *true*, and since the jump vertex's *true*-successor is the target of the jump, it is clear that for every vertex v in the standard CFG G and every initial state σ, the behavior at v when G is executed on σ is the same as the behavior at the corresponding vertex when the augmented CFG is executed on σ.

```
begin
  sum := 0
  prod := 1
  i := 0
  while i < N do
    i := i+1
    if prod > MAXINT/i then
      i := i−1
      break
    fi
    sum := sum+i
    prod := prod*i
  od
  output(i)
  output(sum)
  output(prod)
end
```



(a) Example program and its augmented CFG



(b) Corresponding PDG

Figure 6.4. The example program from Figure 6.1(a), its augmented CFG, and the corresponding PDG. Shading is used to indicate the PDG vertices identified by backwards-closure with respect to "**output**(*prod*)". Flow edges that are not relevant to the backwards-closure with respect to "**output**(*prod*)" are omitted.

Note that in this PDG, the **break** vertex has three outgoing control dependence edges (which are not in the PDG of Figure 6.1(c)). These edges are consistent with the intuition behind control dependence: Removing the **break** might change the number of times the assignments to *sum* and *prod* as well as the evaluation of the loop predicate were performed (and therefore there are control dependence edges from the **break** vertex to the vertices that represent these three components). However, the presence or absence of the **break** has no effect on whether or not statements outside the loop are executed (and therefore there are no control dependence edges from the **break** vertex to a vertex that represents a statement outside the loop).

In Figure 6.4(b), shading is used to indicate the PDG vertices that are identified by backwards-closure with respect to "**output**(*prod*)". Note that the shaded vertices correspond to the program components that are included in the correct program projection shown in Figure 6.1(e). □

We note that if flow dependences are computed in the augmented CFG, some flow dependences arise as a result of paths including dummy edges. Since dummy edges are never actually traversed in an execution, these flow dependences can never be realized in an execution. To avoid these flow dependence edges (and increase the precision of slicing), dummy edges should be ignored when computing flow dependences in the augmented CFG. (Equivalently, flow dependences can be computed in the standard CFG rather than the augmented CFG.)

6.1.2. Sketch of Correctness

In this section we sketch a proof that our slicing algorithm produces a program projection with the desired semantic property.

6.1.2.1. A semantics-preserving transformation on CFGs

The first step of the proof is to show that eliminating the vertices not identified by backwards-closure in the PDG with respect to the slicing vertex (Step 1 of the algorithm) is a semantics-preserving transformation on CFGs.[2] In particular, we show that every vertex in the resulting CFG has equivalent behavior to its corresponding vertex in the original CFG. This part of the proof does not rely at all on the augmented translation. That is, the results here are for arbitrary CFGs, regardless of the program from which they were derived.

---

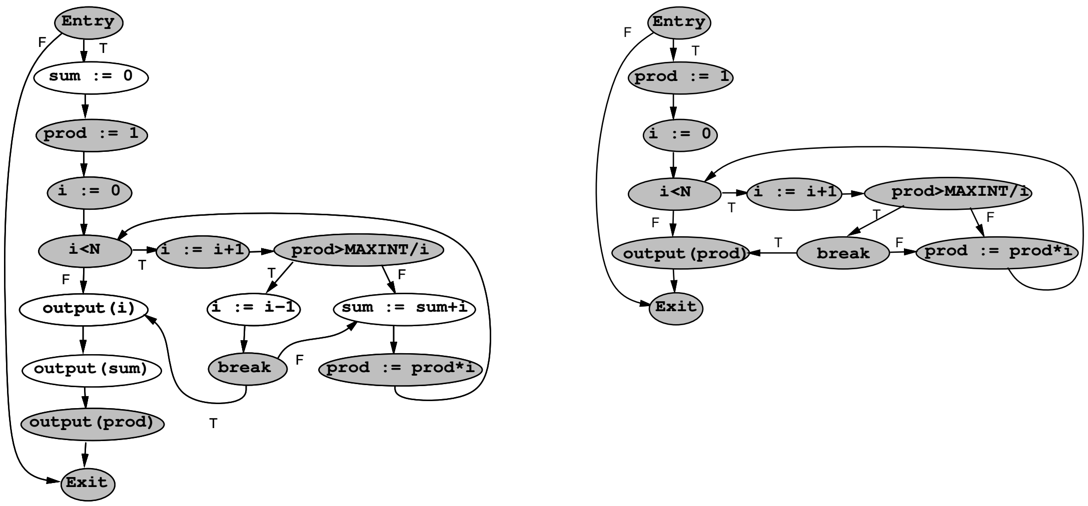
[2] To eliminate a vertex x from CFG G: For every vertex a such that there is an edge $a \to^{L1} x$ and for every vertex b such that there is an edge $x \to^{L2} b$, remove edges $a \to^{L1} x$ and $x \to^{L2} b$; add edge $a \to^{L1} b$. Remove vertex x.

Example. Figure 6.5 repeats the (augmented) CFG of Figure 6.4(a) and shows the CFG that results from eliminating the vertices not identified by backwards-closure with respect to "**output**(*prod*)". □

The proof that eliminating the vertices not identified by backwards-closure in the PDG is a semantics-preserving transformation relies on the following definitions and theorems:

DEFINITION (CFG path). For the purposes of this chapter, we define a path in a CFG as a sequence of alternating vertices and edge labels of the form $(v_1, l_1, \cdots v_{n-1}, l_{n-1}, v_n)$ such that for every label l_i there is an edge $v_i \rightarrow^{l_i} v_{i+1}$ in the CFG.

DEFINITION (project operator). Given a path *PTH* and a set of vertices *V*, project(*PTH*,*V*) is defined to be the sequence resulting from deleting from *PTH* each vertex v_i and label l_i such that $v_i \notin V$. We refer to *PTH* as a *generating path* of project(*PTH*, *V*).



(a) Original (augmented) CFG of the example program. Shading indicates the vertices identified by backwards-closure with respect to "**output**(*prod*)".

(b) After eliminating unshaded vertices.

Figure 6.5. Eliminating vertices not in the backwards-closure preserves CFG semantics.

DEFINITION (path-projection). CFG H is a *path-projection* of CFG G iff all of the following hold:

(1) $V(H) \subseteq V(G)$, the vertices of H are a subset of the vertices of G.[3]

(2) For each path PTH in G, project(PTH, $V(H)$) is a path in H.

(3) For each path in H there is a generating path in G for that path.

DEFINITION (flow/path-projection). CFG H is a *flow/path-projection* of CFG G iff both of the following hold:

(1) H is a path-projection of G.

(2) For every vertex $w \in V(H)$, if G induces the flow dependence $v \longrightarrow_f w$, then $v \in V(H)$.

Example. Figure 6.6 shows four CFGs. Both H and J are path-projections of G; however, K is not. This is because G includes the path (Entry, T, $x>0$, F, $y:=0$, *null*, **output**(y), *null*, **output**(x), *null*, Exit), but the path (Entry, T, Exit) is not in K. H is also a flow/path-projection of G, but J is not. This is because vertex "**output**(y)" is in J, graph G induces a flow dependence from "$y:=1$" to "**output**(y)", but vertex "$y:=1$" is not in J. □

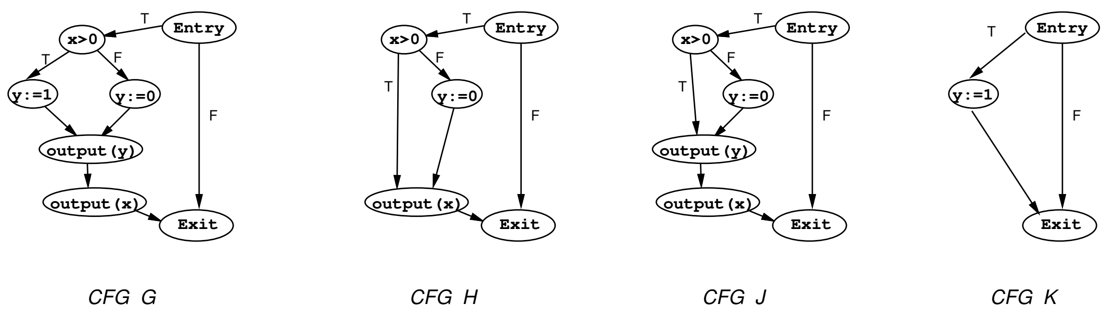


Figure 6.6. H and J are path-projections of G; K is not. H is also a flow/path-projection of G; J is not.

[3]Given a graph G, $V(G)$ denotes the vertex set of G. We use this notation to clarify which graph is under consideration.

THEOREM 6.1. (flow/path-projections preserve CFG semantics). If CFG H is a flow/path-projection of CFG G, then for each vertex $v \in V(H)$:

- For all σ such that $G(\sigma)$ terminates normally, $H(\sigma)$ terminates normally and $G(\sigma)(v) = H(\sigma)(v)$.
- For all σ such that $G(\sigma)$ does not terminate normally, $G(\sigma)(v)$ is a prefix of $H(\sigma)(v)$.

It is straightforward to see that these two points imply that each vertex in CFG H has equivalent behavior to its corresponding vertex in CFG G.

PROOF. See Section 6.2.1.

The following theorem shows that backwards-closure in the PDG finds the set of vertices necessary *and* sufficient to form a CFG that is a flow/path-projection of the original CFG.

THEOREM 6.2. (backwards closure in PDG finds flow/path-projection). Given: CFG G, its PDG D, and vertex v. Eliminating the vertices from G that are not in the backwards closure of D with respect to v yields a CFG that is the minimal flow/path-projection of G that contains v.

PROOF. See Section 6.2.2.

6.1.2.2. A semantics-preserving transformation on programs

Recall that the goal of program slicing is to produce a projection of a given *program*, not to produce a projection of a given CFG. As illustrated by the example of Figure 6.1, under the standard control-flow translation, creating a program projection by eliminating components that do not correspond to the vertices identified by backwards-closure does *not* result in a projection with the desired semantic property.

The second part of the proof of correctness of our algorithm involves showing that under the augmented translation, eliminating program components that do not correspond to the vertices identified by backwards-closure (Step 2 of the algorithm) *is* a semantics-preserving transformation on programs. To prove this, we show that the relationships pictured in Figure 6.7(a) hold. That is, given a program P and a component c, we show that the program Q that results from applying our slicing algorithm to P has a CFG H that is a flow/path-projection of P's CFG G. By

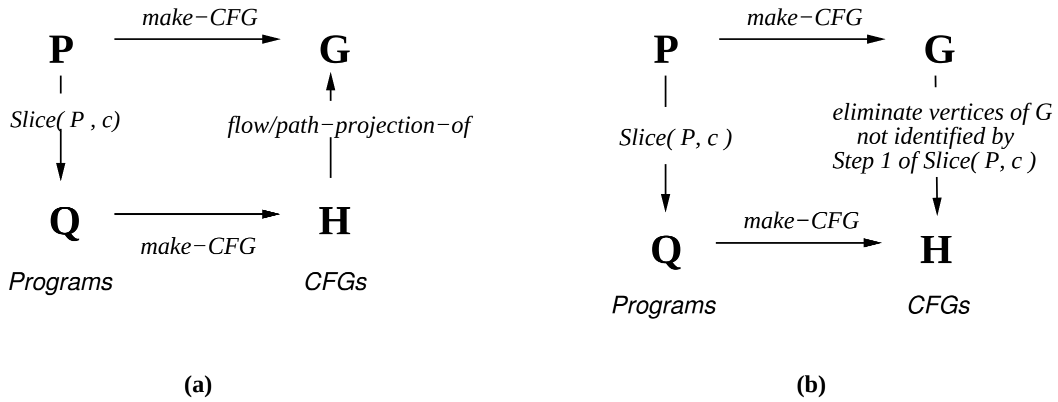


Figure 6.7. Slicing is a semantics-preserving transformation on programs.

the results of the previous section, this means that the vertices in H (in particular, the vertex that corresponds to c) have equivalent behavior to their corresponding vertices in G. This part of the proof relies on several properties relating the abstract syntax of the language to the augmented CFG. Rather than argue directly that H is a flow/path-projection of G, we have shown that H can be obtained from G by eliminating all vertices not identified by backwards-closure, as pictured in Figure 6.7(b). It then follows from the results of the previous section that H is a flow/path-projection of G. See Section 6.2.3 for the proof.

6.2. PROOF OF CORRECTNESS

This section contains proofs of the results stated in Section 6.1.2.

6.2.1. The Behavior of Flow/Path-Projections

This section proves Theorem 6.1. The *execution path* of execution $G(\sigma)$ is the (possibly infinite) path executed by $G(\sigma)$. If the execution terminates normally, the path ends with the *EXIT* vertex. Otherwise, the path is infinite or ends at the first point of failure (*i.e.*, a vertex at which an exception occurs). The i^{th} *instance* of a vertex v in $G(\sigma)$, denoted by v^i, is the i^{th} occurrence of vertex v in the execution path of $G(\sigma)$.

The proof of Theorem 6.1 follows directly from the following lemma. This lemma shows that if H is a flow/path-projection of G, then, for any initial state σ, the execution path of $H(\sigma)$ is a projection of the execution path of $G(\sigma)$ and, furthermore, that intermediate states at corresponding instances in the two executions agree on certain variables. Since H is a path-projection of G, it may omit some assignment statements that appear in G. Therefore, we cannot expect the intermediate states of $H(\sigma)$ and $G(\sigma)$ to agree on all variables. Instead, we show that the intermediate states before an instance v^i in $H(\sigma)$ and its corresponding instance in $G(\sigma)$ are guaranteed to agree on all variables that are live before vertex v in CFG H.

DEFINITION (state at execution of instance v_i). state($G(\sigma)$, v^i) denotes the state immediately before the execution of instance v^i in $G(\sigma)$.

DEFINITION (variables live before vertex v). live_before(G,v) = { x | there is a path in G from v to a vertex w that uses variable x such that no vertex in the path (excluding w but including v) contains an assignment to x }.[4]

LEMMA 6.3. If CFG H is a flow/path-projection of CFG G then for any initial state σ, if *PTH* is a prefix of $G(\sigma)$'s execution path then

(1) project(*PTH*, $V(H)$) is a prefix of $H(\sigma)$'s execution path, and

(2) for every instance v^i in *PTH* such that $v \in V(H)$, $\forall x \in$ live_before(H,v):

state($G(\sigma)$,v^i)(x) = state($H(\sigma)$,v^i)(x).

PROOF. By induction on n, the number of vertices in *PTH* that are in $V(H)$.

Base Case: n = 1. In this case, *PTH* = (*ENTRY*). Trivial.

Induction Step: Assume that the lemma holds when *PTH* includes n vertices that are in $V(H)$. Show that the lemma holds when *PTH* includes $n+1$ vertices that are in $V(H)$. Let v^i be the n^{th}

---

[4]We can also add the restriction that no edge in the path is a dummy edge (as done for the computation of flow dependences). This does not affect the correctness of the proof, since a dummy edge can never appear in an execution path.

vertex in *PTH* that is in $V(H)$; let w^j be the $n+1^{st}$ (and last) vertex in *PTH* that is in $V(H)$. *PTH* contains three subpaths of interest (see Figure 6.8(a)):

- *PTH1* is the prefix of *PTH* from *ENTRY* up to and including v^i.
- *PTH2* is the middle part of *PTH* that includes everything in *PTH* from v_i to w^j.
- *PTH3* is the suffix of *PTH* that includes everything from w_j to the end of *PTH*. Note that w_j is the only vertex in *PTH3* that is in $V(H)$.

By point (1) of the Induction Hypothesis, project(*PTH1*, $V(H)$) is a prefix of $H(\sigma)$'s execution path. By point (2), $\forall x \in$ live_before(H,v), state($G(\sigma)$,v^i)(x) = state($H(\sigma)$,v^i)(x). Every variable used at v^i is in live_before(H,v), so the value computed at v^i is the same in the two executions. Since H is a path-projection of G, if v is a fall-through vertex, then w will be the next

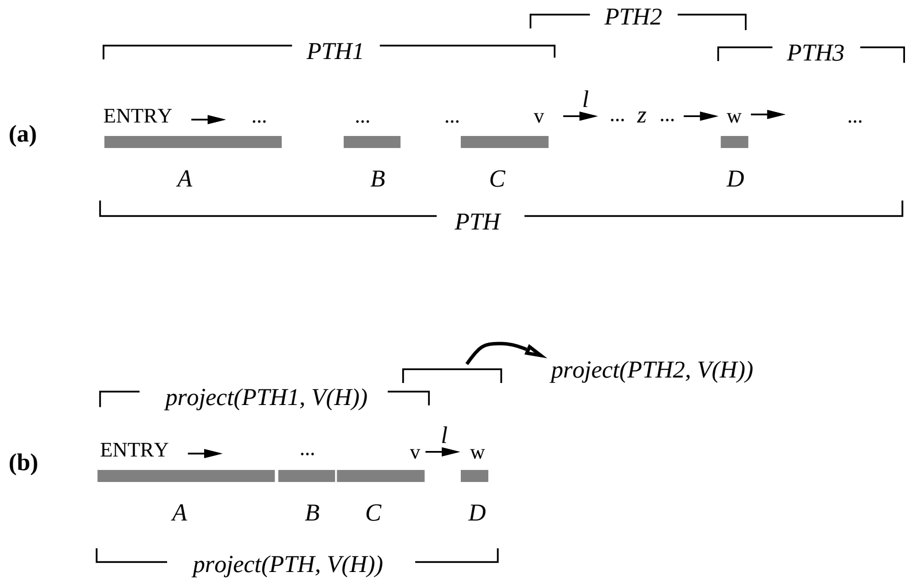


Figure 6.8. An execution path in CFG G (a) and its projection in CFG H (b). The grey bars denote those vertices in the path in CFG G that are in *V(H)*.

vertex to execute after v^i in $H(\sigma)$. If v is a predicate vertex, then since the value computed at v^i is the same in $H(\sigma)$ as in $G(\sigma)$, w will be the next vertex to execute after v^i in $H(\sigma)$. Therefore, project(PTH, $V(H)$) is a prefix of $H(\sigma)$'s execution path (see Figure 6.8(b)).

We now show that $\forall x \in$ live_before(H,w): state($G(\sigma),w^j$)(x) = state($H(\sigma),w^j$)(x). Suppose that $\exists x \in$ live_before(H,w) such that state($G(\sigma),w^j$)(x) $\neq$ state($H(\sigma),w^j$)(x). Since $x \in$ live_before(H,w), either $x \in$ live_before(H,v) and v does not assign to x, or v assigns to x. By the Induction Hypothesis, if $x \in$ live_before(H,v), state($G(\sigma),v^i$)(x) = state($H(\sigma),v^i$)(x). Furthermore, the difference in the value of x at w^j cannot have been caused by an assignment at v, since the same value is computed at v^i in both executions. Therefore, there must be a vertex z that occurs in *PTH2* between v^i and w^j such that: (a) z assigns to x, and (b) there is no other vertex that assigns to x between z and w^j. Let PTH_H be a path in H from w to vertex y that induces x's membership in live_before(H,w) (y uses variable x and no vertex in PTH_H, except possibly y, assigns to x). Let PTH_G be a generating path for PTH_H. Without loss of generality, assume that PTH_G begins with w and ends with y. Since H is a flow/path-projection of G, no vertex in PTH_G, except y, can assign to x (otherwise by point (2) of the definition of flow/path-projection, that vertex would also be in $V(H)$, and by point (2) of the definition of path-projection, it would occur in PTH_H before y). This implies that G induces the flow dependence $z \longrightarrow_f y$. Since $y \in V(H)$ and H is a flow/path-projection of G, z must be a member of $V(H)$. Contradiction. □

PROOF. (Theorem 6.1).

Let CFG H be a flow/path-projection of CFG G. Let PTH be a prefix of $G(\sigma)$'s execution path. By point (1) of lemma 6.3, project($PTH,V(H)$) is a prefix of $H(\sigma)$'s execution path. Point (2) implies that the value of the expressions in corresponding instances v^i in PTH and project($PTH,V(H)$) are the same. We make the following observations:

- If $G(\sigma)$ terminates normally then its execution path is finite and ends with *EXIT*, so $H(\sigma)$ must terminate normally. In this case, for all $v \in V(H)$, $H(\sigma)(v) = G(\sigma)(v)$.
- If $G(\sigma)$ does not terminate normally, then $H(\sigma)$ may or may not terminate normally, depending on whether or not the non-terminating or exception-producing computation in G

is present in H. In either case, for all $v \in V(H)$, $G(\sigma)(v)$ is a prefix of $H(\sigma)(v)$. $\square$

6.2.2. A Semantics-preserving Operation on CFGs

This section proves Theorem 6.2. That is, eliminating from a CFG G the vertices not identified by backwards-closure in the PDG (with respect to a vertex v) produces a CFG H that is a flow/path-projection of G (and includes vertex v). In fact, we only need to argue that H is a path-projection of CFG G. Backwards-closure guarantees that if a vertex w is included in H then all of w's flow dependence predecessors in G's PDG are also in H. Therefore, if H is a path-projection of G, it is guaranteed to be a flow/path-projection too. We also show that CFG H is the minimal flow/path-projection that contains the vertex v; that is, every flow/path-projection of G that contains v must also contain all vertices in the backwards-closure of G's PDG with respect to v.

The set of vertices S identified by backwards-closure (Step 1 of the algorithm) has the property that if w is in S and $v \longrightarrow_c w$ in CFG G's PDG, then v is in S. The following lemma shows that for any set of vertices that satisfies this property, eliminating the vertices not in S from G yields a graph H that is a CFG and a path-projection of G.

LEMMA 6.4. Let G be a CFG and let S be a set of vertices in G such that if $w \in S$ and $v \longrightarrow_c w$ is in G's PDG, then $v \in S$. Eliminating the vertices not in S from CFG G yields a graph H that is a CFG and a path-projection of G.

PROOF. The vertex elimination operation has two properties that are trivial to prove: first, the order in which the vertex elimination operations are applied does not affect the resulting graph; second, if G is a CFG and vertex elimination is applied to some vertex (other than *ENTRY* and *EXIT*) then the resulting graph is a path-projection of G and meets the reachability requirements of a CFG. The problem is that a single application of the vertex elimination operation is not guaranteed to produce a graph that is a CFG. This happens because the operation may create a graph that contains a vertex with two distinct L-successors. For example, consider the graph that results from eliminating the vertex $(x>0)$ from CFG G in Figure 6.6: the *ENTRY* vertex in this graph has two T-successors, $(y:=1)$ and $(y:=0)$.

To complete the proof we must show that there is no vertex with two distinct L-successors in H. The proof is by contradiction. Suppose H contains a vertex v with distinct L-successors y and z. Since H is formed from G by eliminating vertices of G, H is a path-projection of G, as discussed before. Let P_1 be a generating path in G for $v \rightarrow^L y$ and let P_2 be a generating path in G for $v \rightarrow^L z$. Both P_1 and P_2 begin with vertex v, edge label L, and the vertex w that is v's L-successor in G. No vertex in P_1 other than v and y is in $V(H)$; no vertex in P_2 other than v and z is in $V(H)$. It is impossible for both y and z to postdominate each other in G. Without loss of generality, assume that z does not postdominate y in G. Since there is a path in G from w to y and z does not postdominate y, there must be at least one vertex on the path from w to z that is not postdominated by z. Let a be the last such vertex (other than z itself) in this path. Let b be the vertex after a in this path. It is clear that either $z = b$ or z **pd** b. By the definition of control dependence it must be that $a \longrightarrow_c z$, implying that $a \in S$. However, the only vertex in P_2 that is in S is z and $z \neq a$. Contradiction. □

The next lemma shows that backwards closure over control dependence is necessary for creating CFG path-projections. That is, if CFG H is a path-projection of CFG G, $w \in V(H)$ and G induces $v \longrightarrow_c w$, then $v \in V(H)$.

LEMMA 6.5. If CFG H is a path-projection of CFG G, G induces $v \longrightarrow_c^L w$ and $w \in V(H)$, then $v \in V(H)$.

PROOF. Suppose that G induces $v \longrightarrow_c^L w$, $w \in V(H)$, and $v \notin V(H)$. Let P_1 be a path in G from *ENTRY* to v. Let z be the last vertex in P_1 that is in $V(H)$ ($z \neq v$ since $v \notin V(H)$). Let L' be the label on the outgoing edge from z in P_1. Because $v \longrightarrow_c^L w$, the following two paths exist in G: P_2, a w-free path from v's non-L-successor to *EXIT*; P_3, an acyclic path from v's L-successor to w. Since w postdominates the L-branch of v, w postdominates every vertex in P_3 (except itself). It is clear that w cannot postdominate any vertex in P_2. Therefore, P_2 and P_3 have no vertices in common. Since H is a path-projection of G and z is the last vertex in P_1 that is in $V(H)$, z must have an L'-successor in project(P_2,$V(H)$)— since *EXIT* is in $V(H)$—and an L'-successor in project($P_3, V(H)$)—since w is in $V(H)$. Since P_2 and P_3 have no vertices in common, these two vertices must be distinct, which means H is not a CFG. Contradiction. □

6.2.3. A Semantics-preserving Operation on Programs

The results of this section rely on the augmented control-flow translation. In the remainder of this section, we use the word "CFG" to mean "augmented CFG".

This section shows Step 2 of our slicing algorithm, eliminating *stmt* subtrees that do not correspond to the vertices identified by Step 1 (backwards-closure), is a semantics-preserving transformation on *programs*. To prove this, we show that given a program P and a component c, the program Q that results from applying our slicing algorithm to P has a CFG H that is a flow/path-projection of P's CFG G. Rather than arguing directly that H is a flow/path-projection of G, we show that H is identical to the CFG obtained from G by eliminating all vertices not identified by backwards-closure. The results of the previous section imply that this CFG is a flow/path-projection of G.

The proof of this result focuses on the relationship between transitive control dependence and a program's abstract syntax tree. The proof has three main parts:

(1) We first show that the CFG vertices generated by productions in *stmt* subtree T are a subset of transCD($G, vert(T)$). This implies that when the slicing algorithm eliminates a *stmt* subtree (because $vert(T)$ is not in the set S identified by Step 1) it does not eliminate any vertices that are in S. (Section 6.2.3.1).

(2) We next show that eliminating the subtrees (see Figure 6.2) from program P that correspond to vertices in transCD(G, v) yields a program Q with CFG H that is identical to the CFG resulting from eliminating the vertices in transCD(G, v) from G (Section 6.2.3.2). Lemma 6.4 guarantees that eliminating the vertices in transCD(G, v) from CFG G produces a CFG. The edge set of H is:

$$\{\, y \to^L z \mid \text{neither } y \text{ nor } z \text{ is in transCD}(G,v) \text{ and } y \to^L z \text{ is in CFG } G \,\}$$
$$\cup \quad \{\, y \to^L \text{ipd}(G,v) \mid y \notin \text{transCD}(G,v), \text{ and } \exists z \in \text{transCD}(G,v) \text{ such that } y \to^L z \text{ is in CFG } G \,\}$$

Figure 6.9 illustrates the effect of eliminating the vertices in transCD(G,v) from CFG G.

(3) Let S be the set of vertices identified by backwards-closure in the PDG. Eliminating the vertices in $V(G)$–S corresponds to eliminating multiple transCD sets rather than just one transCD set (as in point (2)). Using the above two results, we show the main result of this section: eliminating the subtrees from program P that correspond to vertices in $V(G)$–S yields a program whose CFG is identical to the CFG obtained by eliminating the vertices in $V(G)$–S from G (Section 6.2.3.3).

6.2.3.1. The relationship between transCD and *stmt* subtrees

Consider any *stmt* subtree T in a program and the program's CFG. Let $Verts(T)$ be the set of CFG vertices defined by the productions in *stmt* subtree T. In the augmented translation, every vertex in $Verts(T)$ is reachable from $vert(T)$ and no vertex in $Verts(T)$ postdominates $vert(T)$

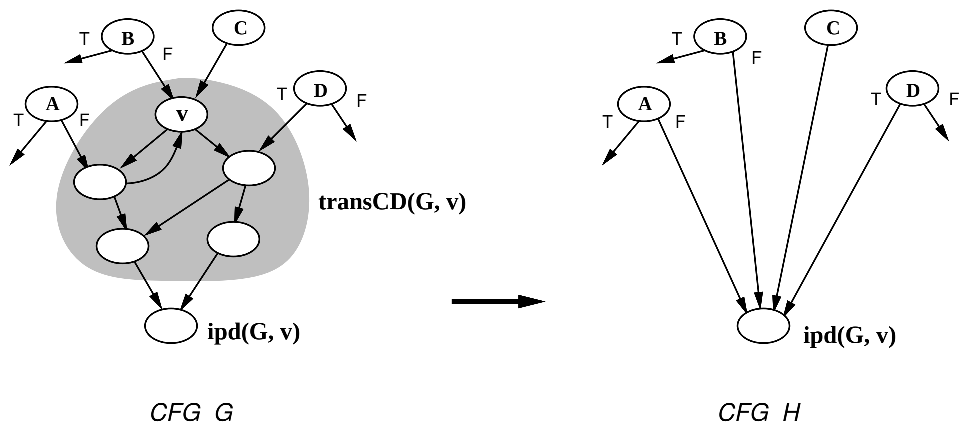


Figure 6.9. Eliminating the vertices in transCD(G,v).

(this is clearly true for any *stmt* subtree that does not contain a jump statement, even under the standard translation; because jump statements generate an edge to their continuation as well as to their target in the augmented translation, it is also true for subtrees containing jump statements). Therefore, $Verts(T) \subseteq$ transCD($G, vert(T)$). However, since unconditional jumps may transfer control to any place in the program, transCD($G, vert(T)$) may contain other vertices. It is clear that if $w \in$ transCD(G, v) then transCD(G, w) $\subseteq$ transCD(G, v). Therefore, if S is a *stmt* subtree and $vert(S) \in$ transCD($G, vert(T)$) then

$$Verts(S) \subseteq \text{transCD}(G, vert(S)) \subseteq \text{transCD}(G, vert(T)).$$

This implies that any transCD(G, v) can be expressed as the union of the *Verts* sets of a set of subtrees. We say that a *stmt* subtree T is contained in a set of vertices V iff $Verts(T) \neq \emptyset$ and $Verts(T) \subseteq V$.

6.2.3.2. Vertex and subtree elimination commute for transCD

Let P be a program with CFG G. Let P_i be the program with CFG G_i resulting from eliminating i subtrees in transCD(G, v) from program P. We show (by induction) that the following invariant holds for all i: the edge set of G_i minus the set { $y \to z$ | $y \in$ transCD(G, v) and $z \in$ transCD(G, v) $\cup$ ipd(G, v) } is

$$\{\, y \to^L z \mid \text{neither } y \text{ nor } z \text{ is in transCD}(G,v) \text{ and } y \to^L z \text{ is in CFG } G \,\}$$
$$\cup \quad \{\, y \to^L z \mid y \notin \text{transCD}(G,v),\ z \in \text{transCD}(G,v) \cup \text{ipd}(G,v),$$
$$\text{and } \exists z' \in \text{transCD}(G,v) \text{ such that } y \to^L z' \text{ is in CFG } G) \,\}$$

Let Q be the program resulting from eliminating all subtrees in transCD(G, v) from program P. Since all the subtrees contained in transCD(G, v) have been eliminated, the CFG of program Q does not contain any of the vertices in transCD(G, v). By the invariant, the CFG of program Q is:

$$\{\, y \to^L z \mid \text{neither } y \text{ nor } z \text{ is in transCD}(G,v) \text{ and } y \to^L z \text{ is in CFG } G \,\}$$
$$\cup \quad \{\, y \to^L \text{ipd}(G,v) \mid y \notin \text{transCD}(G,v) \text{ and } \exists z \in \text{transCD}(G,v) \text{ such that}$$
$$y \to^L z \text{ is in CFG } G \,\}$$

which is exactly the graph that results from eliminating all vertices in transCD(G,v) from CFG G. The proof of the above invariant relies on the following lemma, which characterizes how the elimination of a single *stmt* subtree from a program affects control-flow. Let $T_P.attr$ denote the value of the attribute *attr* in the root production of subtree T in program P.

LEMMA 6.6. Let P be a program with CFG G. Let Q be the program resulting from eliminating *stmt* subtree T from program P. The edge set of program Q's CFG is:

$$\{\ y \rightarrow^L z \mid \text{neither } y \text{ nor } z \text{ is in } Verts(T) \text{ and } y \rightarrow^L z \text{ is in CFG } G\ \}$$
$$\cup \quad \{\ y \rightarrow^L T_P.cont \mid y \notin Verts(T) \text{ and } \exists z \in Verts(T) \text{ such that } y \rightarrow^L z \text{ is in CFG } G\ \}$$

PROOF. Subtree T is replaced by the **NullStmt** subtree (or a sequence of **Label** subtrees). Note that $T_Q.entry = T_Q.cont = T_P.cont$. The proof of this lemma follows from the following observations: any attribute in program P whose value is a vertex in $Verts(T)$ has the value $T_P.cont$ in program Q; any attribute in program P whose value is a vertex not in $Verts(T)$ has the same value in program Q. □

We are now in a position to prove the invariant. Let P be the original program with CFG G. The proof is by induction on the number of subtrees in transCD(G,v) that have been eliminated from program P.

Base Case: No subtrees eliminated. The invariant clearly holds.

Induction Step: Suppose that the invariant is true after the elimination of n subtrees contained in transCD(G,v) from program P. Let P' be the resulting program with CFG G'. Let T be a *stmt* subtree in transCD(G,v) that is in program P' and let Q be the program (with CFG H) resulting from eliminating T from P'. By lemma 6.6, the edge set of H is:

$$\{\ y \rightarrow^L z \mid \text{neither } y \text{ nor } z \text{ is in } Verts(T) \text{ and } y \rightarrow^L z \text{ is in CFG } G'\ \}$$
$$\cup \quad \{\ y \rightarrow^L T_{P'}.cont \mid y \notin Verts(T) \text{ and } \exists z \in Verts(T) \text{ such that } y \rightarrow^L z \text{ is in CFG } G'\ \}$$

If we can show that $T_{P'}.cont \in$ transCD(G,v) $\cup$ ipd(G,v) then CFG H satisfies the invariant (since $Verts(T) \subseteq$ transCD(G,v) and CFG G' satisfies the invariant). To see that $T_{P'}.cont \in$ transCD(G,v) $\cup$ ipd(G,v), note that there must be a vertex $y \in Verts(T)$ such that $y \rightarrow T_{P'}.cont$ is in CFG G' (this is clearly true if T does not contain a jump statement, even under the standard translation; because jump statements generate an edge to their continuation as well as to their target in the augmented translation, it is also true if T contains jump statements). Since the invariant was assumed to hold for G', and since $y \in$ transCD(G,v), it must be that $T_{P'}.cont$ is in transCD(G,v) $\cup$ ipd(G,v). $\square$

6.2.3.3. Putting it all together

Let S be the set of CFG vertices identified by backwards-closure in the PDG. Let $V = V(G) - S$. The set of vertices V is closed under transCD (*i.e.*, for all $v \in V$, transCD(G,v) $\subseteq V$). Eliminating the vertices in V corresponds to eliminating a set of transCD sets, which corresponds to a set of subtrees in program P. Let Q be the program (with CFG H) resulting from eliminating the subtrees associated with V. The following lemma shows (by induction on the size of V) that the CFG H is identical to the CFG obtained by eliminating the vertices in V from CFG G:

LEMMA. Given program P with CFG G and a set of vertices V in G that is closed under transCD. Let Q be the program (with CFG H) that results from eliminating from P all subtrees that correspond to vertices in V. CFG H is identical to the CFG obtained by eliminating the vertices in V from G.

PROOF. By induction on the size of V.

Base Case: $|V| = 1$. That is, $V = \{v\}$, so transCD(G,v) $= \{v\}$. The result of the previous section implies that CFG H is the CFG resulting from eliminating v from CFG G.

Induction Step: Assume that the result is true for V of size less than n. Suppose that V is of size n. Let v be a vertex in V. Let P' be the program resulting from eliminating the subtrees in transCD(G,v) from program P, and let G' be the CFG of P'. Let $V' = V -$ transCD(G,v). V' is clearly of size less than n. To apply the induction hypothesis, we must show that V' is closed under transCD.

By the results of the previous section, G' is the CFG resulting from eliminating the vertices in transCD(G,v) from CFG G. It can be shown (see lemma below) that for any vertex w in G', transCD(G',w) = transCD(G,w) − transCD(G,v). This fact implies the following: (1) for every vertex $w \in V'$, transCD(G',w) $\subseteq V'$. Therefore, V' is closed under transCD; (2) the subtrees of P' contained in V' are the subtrees of P contained in V minus the subtrees of P contained in transCD(G,v); By the Induction Hypothesis, eliminating the subtrees in V' from P' yields a program Q whose CFG H is identical to the CFG resulting from eliminating the vertices in V' from G'. Program Q is the program that results from eliminating all subtrees in V from program P and CFG H is the CFG that results from eliminating the vertices in V from G. This proves our main result. □

LEMMA. Let G be a CFG and let G' be the CFG resulting from eliminating the vertices in transCD(G,v) from G. For any vertex w in G', transCD(G',w) = transCD(G,w) − transCD(G,v).

PROOF. Since G' is a path-projection of G, for any distinct pair of vertices (y,z) in G', y **pd** z in G' iff y **pd** z in G.

Any path $PTH_G{}'$ in G' that starts with w and contains no postdominators of w contributes all its vertices to transCD(G',w). Any generating path in G for PTH_G must contribute the same vertices to transCD(G,w). None of these vertices are in transCD(G,v). Therefore, transCD(G',w) $\subseteq$ transCD(G,w) − transCD(G,v).

Any path PTH_G in G that starts with w and contains no postdominators of w contributes all its vertices to transCD(G,w). Any vertices in the projection of PTH_G in G' must be in transCD(G',w). This projected path does not include vertices from transCD(G,v). Therefore, transCD(G,w) − transCD(G,v) $\subseteq$ transCD(G',w). □

6.3. SLICES AND MINIMALITY

A slicing algorithm identifies a program projection that behaves identically to the original program at some point of interest. As has been noted before, the usefulness of a slicing algorithm is inversely proportional to the size of the slices it produces. While it is an undecidable problem to find slices of minimal size, it would be possible to employ common compiler optimizations to further reduce the size of slices. For example, copy propagation could be used to prune away copy chains from a slice, as shown below (of course, some renaming may need to be done also):

```
x := x+1;          x := x+1;
y := x;
z := y;            z := x;
```

We believe that smaller slices are useful, up to a point. In this chapter, we have shown that our slicing algorithm based on the program dependence graph produces programs whose CFGs are flow/path-projections of the original program's CFG. That is, they preserve paths of the original program (modulo projection) and the flow of values between components. While these properties are useful for proving the semantic results, they also are intuitively appealing. A slice that does not preserve the paths in a program or the flow of values amongst its components may compute the same result as the original program, but does so in a different way than the programmer originally intended. Flow/path-projections retain the *structure* of the computation as well as its result.

We also have shown that backwards-closure in the PDG identifies the minimal set of vertices needed to form a flow/path-projection of a CFG (that includes a given vertex). In particular, control dependence identifies the vertices that must necessarily be included in a slice in order to form a path-projection. That is, if CFG H is a path-projection of CFG G, w is in H and $v \longrightarrow_c w$ is in G's PDG, then v must also be in H.

There are cases where a flow/path-projection that is minimal under the augmented translation is not minimal under the standard translation. In the example below, program Y is the projection that results from slicing program X with respect to C (using the augmented translation). However, under the standard translation, the CFG of program Z is a path-projection of program Y's CFG. Under the standard translation, C is not control dependent on predicate Q (as there is no path from Q to C in the standard CFG). Therefore, Q is not needed in order to form a path-projection including C (under the standard translation). However, under the augmented translation, C is control dependent on Q.

X	*Y*	*Z*
if P **then**	**if** P **then**	**if** P **then**
if Q **then**	**if** Q **then**	
A		
goto L;	goto L;	
fi	**fi**	
B		
goto L;	goto L;	goto L;
fi	**fi**	**fi**
C	C	C
L: D	L:	L:

It remains an open question whether there is an efficient algorithm for finding a program projection that is the minimal flow/path-projection of the original program under the standard control-flow translation.

We have defined the slice of a program to be a projection of that program. That is, the program slice must be formed by eliminating statements from the original program. Because one of the major applications of slicing is debugging, this is a natural restriction. Presenting the programmer with a slice that does not resemble the original program is clearly unsatisfactory. However, if we drop the requirement that the resulting program be a projection of the original program then it is easy to construct programs that are minimal flow/path-projections with respect to the standard control-flow translation. For example, given a program *P* with standard CFG *G*, one could construct the minimal flow/path-projection of *G* with respect to some vertex (using backwards-closure in the PDG to identify the required vertices) and then synthesize a program from that CFG using a structuring algorithm such as Baker's [2]. However, in a language with unstructured control-flow, there can be many programs with the same CFG. The program that results from such an approach may not be a projection of the original program, even though it meets the semantic goal (because its CFG is a flow/path-projection of the original program's CFG).

6.4. OTHER CONTROL CONSTRUCTS

The language considered in this chapter has arbitrary control-flow, due to the inclusion of the **goto** statement. It also has looping and conditional constructs (**repeat** and **while**) found in many languages. However, the question naturally arises: do the results of this chapter extend to other control constructs, such as **for** loops and **case** statements? As we have shown, the program dependence graph can be used to form flow/path-projections of completely arbitrary control-flow graphs. However, to ensure that the program projection operation works (see Figure 6.7) control constructs must satisfy a few simple properties.

A looping construct must generate a vertex v in the control-flow graph such that: (1) v passes control to the continuation of the loop construct (*i.e.*, there is a loop exit); (2) every vertex generated by the (abstract syntax) subtrees enclosed by the looping construct is reachable from v. A **for** loop meets these requirements. However, a construct such as **loop-forever** does not. Fortunately, it is usually possible to translate a construct so that by the addition of dummy vertices and edges, it meets the requirements. For example, a **loop-forever** construct can be treated as a **while** loop where the predicate is *true*. This results in a dummy vertex with an outgoing *false* edge that is not executable.

A selection construct must generate a vertex v such that: (1) every vertex generated by subtrees enclosed by the selection construct is reachable from v; (2) every subtree immediately enclosed by the selection construct passes control to the continuation of the selection construct; The translations of selection constructs such as **if-then**, **if-then-else**, and **case** meet these requirements.

6.5. RELATED WORK

As mentioned previously, Weiser defined the first program slicing algorithm [71]. The Ottensteins defined a more efficient program slicing algorithm using the program dependence graph [52]. Neither algorithm handles unstructured control-flow correctly. Lyle defined an algorithm to slice programs containing **goto**s, which is summarized by Gallagher [21], but it is quite conservative (*i.e.*, produces projections that are larger than necessary), as we show below. Defined in our terminology, Lyle's algorithm includes a **goto** in the projection only if there is a flow dependence $x \longrightarrow_f y$ traversed by the backwards-closure such that the **goto** is in a control-flow path that induces the flow dependence $x \longrightarrow_f y$. Consider slicing the following program with respect to its last statement:

```
x := 1
i := 1
while i < M do
    if i = N then goto L fi
    i := i+1
od
L:  y := x
```

Because the **goto** is in a path that induces the flow dependence from (x:=1) to (y:=x), Lyle's algorithm will include the **goto** and, as a result, the entire **while** loop and the statement (i:=1), in addition to (y:=x) and (x:=1). His algorithm forms a projection that is identical to the original program. However, the following program is an acceptable projection (which our slicing algorithm will produce):

```
x := 1
y := x
```

Gallagher made a refinement to Lyle's algorithm to address this imprecision [21], but his algorithm, as defined in his thesis, may produce semantically incorrect projections. Basically, Gallagher's algorithm includes "**goto** *L*" only if a control dependence predecessor of the **goto** is in the slice and the statement labelled *L* is included in the slice. The following example shows a program for which both Gallagher's algorithm fails to produce a semantically correct projection. Suppose that we wish to slice program *X* with respect to the statement (y:=x+1).

X	*Y*	*Z*
if P **then goto** L **fi** x := 1 y := x+1 L: a := 2	**if** P **then fi** x := 1 y := x+1	**if** P **then goto** L **fi** x := 1 y := x+1 L:

Program *Y* is the projection that will result from the application of Gallagher's algorithm. This program is not a semantically correct projection of program *X* because in program *X*, if the variable P is true then the statements (x:=1) and (y:=x+1) are not executed, while in program *Y* they execute unconditionally. Gallagher's algorithm does not include the **goto** because the statement (a:=2) is not in the slice.

Gallagher has modified his slicing algorithm since the publication of his thesis [22]. However, although his updated algorithm seems to be an improvement over the algorithm given in his thesis, he has no proof that it is correct and there still seem to be cases in which it fails to produce a semantically correct slice.

Reps and Yang gave the first formal proof that the program slices formed by using the program dependence graph have the desired semantic property [62]. Furthermore, they showed that slicing using the program dependence graph guarantees equivalent behavior at every point in the slice (not just at the slicing vertex). However, they proved this only for programs with structured control-flow. We have shown that the program dependence graph can be used to slice programs with arbitrary control-flow with the guarantee of equivalent behavior at every point. We note that Reps and Yang defined a program slice to allow the possible reordering of program statements (including conditionals and loops). Under their definition, the second program shown below is a slice of the first (and vice versa):

```
a := 1          b := 2
b := 2          a := 1
c := a+b        c := a+b
```

Although the ordering of statements in the two programs differs, the same sequence of values is computed at each point. Under our framework, neither program's CFG is a path-projection of the other's CFG, so our semantic result about flow/path-projections cannot be applied to compare the programs' behaviors. However, we believe it is possible to extend slicing with reordering even in the presence of arbitrary control-flow.

Choi and Ferrante independently discovered the same problem of slicing programs with arbitrary control-flow (using the program dependence graph)[12]. They proposed two solutions to the problem. The first uses the idea of an augmented CFG, much the same as ours. The second solution uses the PDG of the program's standard CFG to decide which statements to eliminate. In addition to deleting statements from the original program, their second approach inserts additional **goto**s to ensure correct control-flow. Thus, the resultant program may not be a projection of the original program. As discussed in Section 6.3, if it is not necessary to form a program projection, then the PDG of the standard CFG can be used to form a minimal CFG flow/path-projection (with respect to the standard translation). A structuring algorithm can then be used to form a program from the CFG. Structuring algorithms attempt to minimize the number of **goto**s needed and will probably produce more readable code than the second approach of Choi and Ferrante.

The major difference between our work and the first solution proposed by Choi and Ferrante is the generality of the results. Our algorithm is defined for a language that includes (arbitrarily nested) conditional statements and loops as well as **break**s and **goto**s. In contrast, Choi and Ferrante's first algorithm is defined for a much more limited language in which the only constructs that affect control-flow are conditional and unconditional **goto**s. As Choi and Ferrante note, any structured control construct (such as an **if-then-else** or a **while** loop) can be synthesized in this simple language. However, as the following example shows, synthesizing control constructs in their simple language can lead to unnecessarily larger slices when the augmented CFG is used. Consider the following structured code and its translation into Choi and Ferrante's language:

if P **then**		**if** not(P) **then goto** 1;
A		A;
if Q **then**		**if** Q **then goto** 3;
halt		**goto** 2;
fi	1:	B;
else	2:	C;
B	3:	
fi		
C		

Under the augmented control-flow translation of the first program, there is no path from predicate Q to statement B, so B cannot be control dependent on Q. However, in the second program, statement B is control dependent on Q because of the edge from "**goto** 2" to B. Thus, a slice with respect to B in this program picks up predicate Q. One could argue that the statement "**goto** 2" should not be treated the same as other **goto**s that are explicitly written by the programmer. However, then one must define some other procedure for determining when these implicit **goto**s are needed in a slice.

Chapter 7

DIFFERENCING AND INTEGRATION IN THE PRESENCE OF COMPLEX CONTROL-FLOW

This chapter presents algorithms for computing the semantic difference of control-flow graphs with completely arbitrary control-flow and integration of control-flow graphs with mostly reducible control-flow. The definition of the differencing operation (Section 7.1) is based on our results on slicing from Chapter 6. It is essential to understand the results on differencing before proceeding to the integration algorithm (Section 7.2), in which differencing plays a key role. Section 7.3 contains proofs of the results stated in Sections 7.1 and 7.2. Section 7.4 discusses related work on differencing and integration.

7.1. DIFFERENCING

Given two programs, *Old* and *New*, and a correspondence between vertices of the programs, the difference of *New* with respect to *Old* includes all vertices of *New* that have no corresponding vertex in *Old*, as well as all vertices in *New* whose corresponding vertex in *Old* has inequivalent behavior. Because it is impossible to identify this set exactly, a safe algorithm will also sometimes include a vertex of *New* whose corresponding vertex in *Old* has equivalent behavior.

This section shows how the results on flow/path-projections from Chapter 6 can be used to compare the behavior of corresponding vertices in different CFGs. These CFGs may contain arbitrary control-flow. We present three algorithms for computing the difference of two CFGs, each more efficient than the previous one. Section 7.1.1 shows how the isomorphism of flow/path-projections can be used to compute the difference of two CFGs, and shows how CFG isomorphism may be efficiently determined. Section 7.1.2 shows that it is not necessary to actually construct flow/path-projections to compute the difference. Rather, we can use the program dependence graph to achieve the same result. Section 7.1.3 shows how the efficiency of PDG-based differencing can be improved upon.

7.1.1. Differencing With Flow/Path-Projections

NOTATION. We will use $FPP(G,v)$ to denote the minimal CFG flow/path-projection of CFG G that contains vertex v.

Let *Old* and *New* be CFGs containing corresponding vertices v and w, respectively. Suppose that CFG $H = FPP(Old,v) = FPP(New,w)$.[1] We can use this fact and Theorem 6.1 (from Chapter 6) to relate the execution behavior of vertex v in *Old* to the behavior of w in *New*. That is, vertex v in *Old* has equivalent behavior to vertex w in *New*, as we show now. Keep in mind that because H is isomorphic to $FPP(Old,v)$ and $FPP(New,w)$, for any inital state σ, $H(\sigma)(v) = H(\sigma)(w)$.

- Let σ be a state such that both $Old(\sigma)$ and $New(\sigma)$ terminate normally. Theorem 6.1 implies that $H(\sigma)$ terminates normally, $H(\sigma)(v) = Old(\sigma)(v)$, and $H(\sigma)(w) = New(\sigma)(w)$. Therefore, $Old(\sigma)(v) = New(\sigma)(w)$.
- Let σ be a state such that neither $Old(\sigma)$ nor $New(\sigma)$ terminates normally. Theorem 6.1 implies that $Old(\sigma)(v)$ is a prefix of $H(\sigma)(v)$, and $New(\sigma)(w)$ is a prefix of $H(\sigma)(w)$. It follows that either $Old(\sigma)(v)$ is a prefix of $New(\sigma)(w)$, or vice versa.
- Let σ be a state such that $Old(\sigma)$ terminates normally and $New(\sigma)$ does not terminate normally. In this case, Theorem 6.1 implies that $H(\sigma)(v) = Old(\sigma)(v)$, and $New(\sigma)(w)$ is a prefix of $H(\sigma)(w)$. Therefore, $New(\sigma)(w)$ is a prefix of $Old(\sigma)(v)$.
- Let σ be a state such that $New(\sigma)$ terminates normally and $Old(\sigma)$ does not terminate normally. In this case, Theorem 6.1 implies that $H(\sigma)(w) = New(\sigma)(w)$, and $Old(\sigma)(v)$ is a prefix of $H(\sigma)(v)$. Therefore, $Old(\sigma)(v)$ is a prefix of $New(\sigma)(w)$.

---

[1] In the context of graphs, = denotes isomorphism. Two graphs G_1 and G_2 are isomorphic if and only if there is a 1-to-1, onto mapping f from $V(G_1)$ to $V(G_2)$ such that the following conditions all hold:

- For every vertex v in $V(G_1)$, v and $f(v)$ have the same text.
- For every edge $v \to w$ in G_1, there is an edge $f(v) \to f(w)$ in G_2 with the same label.
- For every edge $v \to w$ in G_2 there is an edge $f^{-1}(v) \to f^{-1}(w)$ in G_1 with the same label.

We use this observation to define the goal of a differencing algorithm: The difference of *New* with respect to *Old*, denoted by Diff(*New*,*Old*), should include all vertices in *New* that have no corresponding vertex in *Old*, as well as all vertices *w* in *New* with corresponding vertex *v* in *Old* such that $FPP(New,v) \neq FPP(Old,w)$.

DEFINITION (the difference of *New* with respect to *Old*).

$$\begin{aligned} \mathrm{Diff}(New,Old) \quad = \quad & \{\, v \in V(New) \mid v \text{ has no corresponding vertex in } Old \,\} \\ \cup \; & \{\, v \in V(New) \mid FPP(New,v) \neq FPP(Old,v) \,\} \end{aligned}$$

While isomorphism of general graphs appears to be a hard problem to answer efficiently [23], isomorphism of CFGs can be determined in linear time because the outgoing edges of each vertex in a CFG are uniquely labelled. Given CFGs *G* and *H*, simply perform a depth-first search on both *G* and *H* (from *ENTRY*) such that the *False* successor of a predicate vertex is always visited before the *True* successor. If *G* and *H* are isomorphic then the map between vertices established by common depth-first numbers is clearly an isomorphism map. If the CFGs are not isomorphic then no numbering of the vertices can be an isomorphism map. It is easy to check whether the CFGs are isomorphic under the map established by the depth-first numberings, answering the isomorphism question.

7.1.2. Differencing With the Program Dependence Graph

With the algorithm outlined above, to check whether a corresponding pair of vertices in two CFGs has equivalent behavior requires backwards-closure in the PDG of each CFG, formation of the CFG flow/path-projections, and the isomorphism check of these CFGs. We now define a more efficient differencing method, which operates solely on the PDG representation and does not require the construction of the flow/path-projections. This method is also better suited to integration, as will become clear in the next section. We require the following definitions:

DEFINITION (ordering edge). Let *a* and *b* be distinct vertices in a CFG. If there is a vertex *p* such that $p \longrightarrow_c^L a$ and $p \longrightarrow_c^L b$, then vertices *a* and *b* must be ordered by the postdomination relation since a and b both postdominate the L-branch of p. If a **pd** b (a postdominates b) then there is an ordering edge $b \longrightarrow_{or} a$. Otherwise, b **pd** a and there is an ordering edge $a \longrightarrow_{or} b$.

NOTATION (program dependence graph with ordering edges). Given a CFG G, let OPDG(G) be G's program dependence graph augmented with ordering edges, as defined above.

NOTATION (subgraph induced by backwards-closure). Let P be a PDG with or without ordering edges and let S be the set of vertices in P from which v is reachable via a path of control and/or flow dependences (*i.e.*, the vertices in the backwards-closure over control and flow dependences in P). The subgraph of P induced by S is denoted by P/v.[2]

We show in Section 7.3 that $FPP(Old,v) = FPP(New,v)$ iff OPDG(Old)/v = OPDG(New)/v. That is, we can test the isomorphism of the flow/path-projections without constructing them by testing the isomorphism of the ordered PDG subgraphs induced by backwards-closure. Therefore, Diff(New,Old) can be redefined as:

DEFINITION (the difference of *New* with respect to *Old*).

$$\begin{aligned}\mathrm{Diff}(New,Old) \quad = \quad & \{\, v \in V(New) \mid v \text{ has no corresponding vertex in } Old \,\} \\ \cup\ & \{\, v \in V(New) \mid \mathrm{OPDG}(New)/v \neq \mathrm{OPDG}(Old)/v \,\}\end{aligned}$$

Isomorphism of ordered program dependence (sub)graphs can be accomplished in linear time (as is done with CFGs) because the control dependence successors of any vertex are totally ordered by the ordering edges. The depth-first search traverses only control dependence and ordering edges, visiting control dependence successors before ordering successors and visiting the *False* control dependence successors of a predicate vertex before the *True* successors. The depth-first number of a vertex is assigned when leaving a vertex (*i.e.*, postorder).

[2]Given a graph G and a set of vertices $S \subseteq V(G)$, the subgraph of G induced by S is (S, $\{\, v \to w \mid v \to w \in E(G), v \in S, \text{and } w \in S \,\}$).

7.1.3. Improving the Efficiency of PDG-based Differencing

Testing the isomorphism of PDG subgraphs for every pair of corresponding vertices can still be expensive, but can be avoided for many vertices, as we now show. The algorithm in Figure 7.1 computes Diff(*New*,*Old*). The justification for this algorithm is straightforward. If vertex *v* in *New* has no corresponding vertex in *Old* or differs in incoming flow or control dependence edges in PDG(*New*) from PDG(*Old*), then OPDG(*Old*)/*v* cannot be isomorphic to OPDG(*New*)/*v*. The set DAV(*New*,*Old*) contains these vertices. Furthermore, for any vertex *w* reachable from *v* in PDG(*New*), OPDG(*Old*)/*w* cannot be isomorphic to OPDG(*New*)/*w*. However, it is still possible that the only reason two PDG subgraphs are not isomorphic is due to a difference in ordering edges. Unfortunately, a difference in incoming ordering edges to a vertex *v* does not imply that OPDG(*New*)/*v* and OPDG(*Old*)/*v* are not isomorphic. Therefore, after the first two steps of the algorithm in Figure 7.1, the algorithm must check, for each vertex $v \in V(New)$ such that $v \notin$ Diff(*New*,*Old*), if OPDG(*Old*)/*v* ≠ OPDG(*New*)/*v*. If the graphs are not isomorphic, then *v* (and all vertices reachable from *v* in PDG(*New*)) are added to Diff(*New*,*Old*).

DAV(*New*,*Old*) :=
    { $v \in V(New)$ | *v* has no corresponding vertex in $V(Old)$ }
  ∪ { $v \in V(New)$ | *v* has a different set of incoming flow or control dependence edges
        in PDG(*New*) than in OPDG(*Old*) }

Diff(*New*,*Old*) :=
    DAV(*New*,*Old*)
  ∪ { $w \in V(New)$ | *w* is reachable in PDG(*New*) from a vertex $v \in$ DAV(*New*,*Old*) }

while $\exists v \in V(New)$ such that $v \notin$ Diff(*New*,*Old*) and OPDG(*Old*)/*v* ≠ OPDG(*New*)/*v* **do**
  Diff(*New*,*Old*) := Diff(*New*,*Old*) ∪ { *w*| *w* is reachable in PDG(*New*) from vertex *v* }
od

Figure 7.1. Algorithm for computing Diff(*New*,*Old*).

7.2. INTEGRATION

We first show how the results on differencing can be used to characterize when a CFG M is an acceptable integration of CFGs A, B, and $BASE$. This formalizes the goal of our integration algorithm, presented in Sections 7.2.1–7.2.4. We require the following definition:

DEFINITION (Equiv(New, Old)). If Old and New are CFGs, then Equiv(New, Old) is defined to be $V(New) - \text{Diff}(New, Old)$. These are the vertices of New that have a corresponding vertex in Old with equivalent behavior.

Let $BASE$, A, and B be CFGs with correspondences between $BASE$ and A, and $BASE$ and B. The goal of our integration algorithm will be to find a CFG M that meets the following structural criteria:

- If $v \in$ Diff($A, BASE$) then there exists a vertex v in M such that OPDG(M)/v = OPDG(A)/v.[3]
- If $v \in$ Diff($B, BASE$) then there exists a vertex v in M such that OPDG(M)/v = OPDG(B)/v.
- If $v \in$ Equiv($A, BASE$) $\cap$ Equiv($B, BASE$) then there exists a vertex v in M such that OPDG(M)/v = OPDG($BASE$)/v.

In addition, we require that every flow/path-projection of M be a flow/path-projection of CFG A or B. That is, for every vertex v in M, either OPDG(M)/v = OPDG(A)/v, OPDG(M)/v = OPDG(B)/v, or OPDG(M)/v = OPDG($BASE$)/v. This prevents M from containing “spurious” code that could affect whether or not M terminates normally.

We now show that if such a CFG M exists, then it meets the semantic criteria of integration, as defined in Chapter 5. First, we must show that if σ is a state on which A, B, and $BASE$ all terminate normally, then $M(\sigma)$ terminates normally. We show that for any vertex v in M, $M(\sigma)(v)$ is finite and a prefix of either $A(\sigma)(v)$ or $B(\sigma)(v)$. This implies that $M(\sigma)$ terminates normally. By the definition of M, either OPDG(M)/v = OPDG(A)/v or OPDG(M)/v = OPDG(B)/v. Suppose the former, which implies that vertex v has equivalent behavior in A and M. Since $A(\sigma)$

[3]Equivalently, $FPP(M, v) = FPP(A, v)$.

terminates normally, $M(\sigma)(v)$ is finite and a prefix of $A(\sigma)(v)$. The proof is symmetric in the case that OPDG(M)/v = OPDG(B)/v.

We now prove point (2) of the semantic goal of integration. Suppose that σ is a state on which A, B and $BASE$ all terminate normally. By (1), $M(\sigma)$ terminates normally. Suppose that $A(\sigma)(v) \neq BASE(\sigma)(v)$. Vertex v does not have equivalent behavior in A and $BASE$, so $v \in$ Diff(A,$BASE$). By construction, OPDG(M)/v = OPDG(A)/v, so v has equivalent behavior in M and A. Since M and A both terminate normally on σ, $M(\sigma)(v) = A(\sigma)(v)$. Points (3) and (4) of the semantic goal of integration follow by similar arguments.

Ultimately, we would like an integration algorithm that constructs a CFG M meeting the above structural criteria whenever such a CFG exists. We describe an integration algorithm that, given CFGs $BASE$, A and B, will find a CFG M (that meets the structural criteria) when it exists, for a certain class of CFGs. In particular, $BASE$, A and B must be reducible CFGs whose loops must be in a normal form that we call *while* loops. Section 7.2.1 describes this class of CFGs. Following the integration algorithm of Horwitz, Prins and Reps (the HPR algorithm) [30], our integration algorithm has three main steps (Section 7.4 compares our algorithm to the HPR algorithm):

(1) Determine the vertices of A and B that have different behavior than their corresponding vertices in $BASE$ (using differencing, as defined in the previous section), and create a merged program dependence graph P_M. Specifically: if $v \in$ Diff(A,$BASE$) then P_M contains OPDG(A)/v; if $v \in$ Diff(B,$BASE$) then P_M contains OPDG(B)/v; and if $v \in$ Equiv(A,$BASE$) $\cap$ Equiv(B,$BASE$) then P_M contains OPDG($BASE$)/v (Section 7.2.2).

(2) Determine whether or not A and B interfere with respect to $BASE$ by examination of the graphs P_M, OPDG(A), and OPDG(B). Graph P_M passes the interference test iff for each vertex v in P_M: if $v \in$ Diff(A,$BASE$) then P_M/v = OPDG(A)/v; if $v \in$ Diff(B,$BASE$) then P_M/v = OPDG(B)/v; otherwise $v \in$ Equiv(A,$BASE$) $\cap$ Equiv(B,$BASE$) and P_M/v = OPDG($BASE$)/v. If P_M does not pass the interference test then integration fails (Section 7.2.3).

(3) Find a CFG M such that for each vertex v in OPDG(M), OPDG(M)/v = P_M/v. We refer to this process as *reconstitution*. Reconstitution may fail to find such a CFG, in which case integration fails (Section 7.2.4).

If the above algorithm succeeds then it is clear that CFG M meets the structural criteria defined at the beginning of this section, and is an acceptable integration of CFGs A, B, and $BASE$. As mentioned before, our integration algorithm is only guaranteed to work for a certain class of CFGs (described in the next section). This restriction is due to step (3) of the algorithm, the process of reconstituting a CFG from graph P_M. Steps (1) and (2) work for arbitrary CFGs.

7.2.1. Normal-form CFGs and Other Considerations

This section describes the class of CFGs that our integration algorithm accepts and produces. The first property of normal-form CFGs is that they are reducible. Informally stated, reducibility restricts loops to have a single entry point (although they may have more than one exit point). The concept of domination in a graph is used to define reducibility.

DEFINITION (domination in a rooted graph). Vertex v *dominates* vertex w, denoted by v **dom** w, in a graph rooted at *ENTRY* iff every path from *ENTRY* to w contains v.

DEFINITION (reducibility in a rooted graph). A rooted graph G is *reducible* iff the removal of every edge $v \rightarrow w$ from G such that w **dom** v leaves an acyclic graph. Equivalently, G is reducible iff any depth-first search of G identifies the same set of edges as backedges. There are many other equivalent characterizations of reducibility [1].

The second property of normal-form CFGs is that each loop in the CFG must be in the form of a *while* loop. We use the concept of natural loop [1] to define the structure of a while loop:

DEFINITION (natural loop). The natural loop of a backedge $v \rightarrow w$ is defined to be

$$\text{nat-loop}(v \rightarrow w) = \{ w \} \cup \{ x \mid \text{there is a } w\text{-free path from } x \text{ to } v \}.$$

The natural loop associated with a loop entry w (*i.e.*, a vertex that is the target of one or more backedges), denoted by nat-loop(w), is the union of all nat-loop($v \rightarrow w$), where $v \rightarrow w$ is a backedge. Any two natural loops in a reducible CFG are either disjoint or properly nested.

DEFINITION (while loop). A natural-loop nat-loop(w) is a *while* loop if no vertex in nat-loop(w) postdominates w. Each natural loop in a normal-form CFG must be a while loop.[4]

The integration algorithm also requires that flow dependences be further classified as to whether or not they are carried by a loop and that special *initial definition* vertices are present in the CFG.

DEFINITION (loop-carried flow dependences). A flow dependence $x \longrightarrow_f y$ is carried by loop entry w if x and y are both members of nat-loop(w) and there is a path that induces the flow dependence $x \longrightarrow_f y$ that includes a backedge of the form $v \rightarrow w$. A loop-carried flow dependence is labelled with the loop entry of the outermost loop that carries the dependence.

For every variable x that may be used before being defined (*i.e.*, the CFG contains an x-definition free path from *ENTRY* to a use of x), an *initial definition* vertex w = (x := InitialState(σ, x)) must be added to the CFG. Vertex w becomes the *True* successor of *ENTRY* and the former *True* successor of *ENTRY* becomes the successor of w. Vertex w represents the initial assignment of x's value from the initial execution state σ. The order in which the initial definition vertices are added to the CFG is immaterial.

7.2.2. Construction of the Merged Dependence Graph P_M

This section describes how to construct the merged dependence graph P_M. Given CFGs *BASE*, *A*, and *B* (and the correspondence maps), differencing determines the sets Diff(A,$BASE$), Equiv(A,$BASE$), Diff(B,$BASE$), and Equiv(B,$BASE$). Any vertex that in Diff(A,$BASE$) or Diff(B,$BASE$) represents a possible change of behavior and should be included in P_M. The subgraph (of the relevant PDG) that is responsible for this behavior also is included in P_M. Any

[4]A loop in a reducible CFG that is not a while loop can easily be transformed into one while preserving reducibility and the transitive control dependence relation.

vertex that is in Equiv($A, BASE$) $\cap$ Equiv($B, BASE$) and its associated subgraph should also be included in P_M. Graph P_M is the graph union of three subgraphs (where the correspondence maps identify identical vertices across program dependence graphs):

$$
\begin{aligned}
P_M \quad = \quad & \{ \text{OPDG}(A)/v \mid v \in \text{Diff}(A, BASE) \} \\
\cup \quad & \{ \text{OPDG}(B)/v \mid v \in \text{Diff}(B, BASE) \} \\
\cup \quad & \{ \text{OPDG}(BASE)/v \mid v \in \text{Equiv}(B, BASE) \cap \text{Equiv}(B, BASE) \}
\end{aligned}
$$

There are a few facts about the structure of P_M that are worth pointing out, as they affect the next two steps of the integration algorithm. First, a vertex v may have control, flow, or ordering predecessors in P_M that it does not have in its PDG of origin. This occurs if and only if there is a vertex v in $BASE$ with corresponding vertices in both A and B such that $v \in$ Diff($A, BASE$), $v \in$ Diff($B, BASE$), and OPDG(A)/$v \neq$ OPDG(B)/v. Second, not every pair of vertices with a common control dependence parent in P_M is ordered by an ordering edge (as is the case in OPDG(A), OPDG(B), and OPDG($BASE$)).

7.2.3. Interference Between Variants

By construction, a vertex in the graph P_M is either from Diff($A, BASE$), Diff($B, BASE$), or Equiv($A, BASE$) $\cap$ Equiv($B, BASE$). Variants A and B interfere with one another iff there is a vertex $v \in$ Diff($A, BASE$) such that $P_M/v \neq$ OPDG(A)/v, or there is a vertex $v \in$ Diff($B, BASE$) such that $P_M/v \neq$ OPDG(B)/v. It is clear that for any vertex v in Equiv($A, BASE$) $\cap$ Equiv($B, BASE$), OPDG(A)/v = OPDG(B)/v. Therefore, P_M/v = OPDG(A)/v = OPDG(B)/v.

If there is interference as defined above then the integration algorithm fails. Horwitz, Prins, and Reps call this Type I interference [30]. Reps and Bricker describe how the results of interference can be used to guide the programmer to code in variants A and B that introduces the conflict [61].

7.2.4. Reconstitution of a CFG from P_M

As stated before, the goal of reconstitution is to find a CFG *M* that includes all of the vertices of P_M, such that for every vertex *v* in OPDG(*M*), OPDG(*M*)/*v* = P_M/v. Because P_M is the union of subgraphs from OPDG(*A*), OPDG(*B*) and OPDG(*BASE*), it may be only a *partially* ordered program dependence graph. That is, there may be vertices *a* and *b* that share a common control parent in P_M (i.e., there exists a *p* such that $p \longrightarrow_c^L a$ and $p \longrightarrow_c^L b$), yet there is no ordering edge between *a* and *b* in P_M. This complicates the reconstitution process. To explain why, we need the following definitions:

DEFINITION ((a,b) pair). A distinct pair of vertices (a,b) in a CDG is an "(a,b) pair" if there is a predicate *p* in the CDG such that $p \longrightarrow_c^L a$ and $p \longrightarrow_c^L b$.

DEFINITION (corresponding CFG). Given a (partially) ordered PDG *P*, CFG *M* is a *corresponding CFG of P* iff OPDG(*M*) is a supergraph of *P* that includes the same set of vertices, control edges, and flow edges, and a superset of *P*'s ordering edges.

DEFINITION (feasible PDG). (Partially) ordered PDG *P* is *feasible* iff it has at least one corresponding CFG. Otherwise, *P* is *infeasible*.

DEFINITION (*R*-feasible): (Partially) ordered PDG *P* is *R-feasible* iff it has at least one corresponding normal-form CFG. Otherwise, *P* is *R-infeasible*

Given an arbitrary (partially) ordered PDG *P*, it is not in general true that every corresponding CFG *M* will satisfy the goal of reconstitution (for every vertex *v* in OPDG(*M*), OPDG(*M*)/*v* = P/v). This is because *P* may be missing an ordering edge between two vertices that are both included in P/v. However, in the context of integration, this is *not* possible; that is, every CFG that corresponds to P_M will satisfy the goal of reconstitution. This is because P_M is the graph union of subgraphs of the form OPDG(*A*)/*v*, OPDG(*B*)/*v*, or OPDG(*BASE*)/*v*.

Therefore, reconstituting P_M involves finding a corresponding CFG, or determining that no such CFG exists. This is accomplished by adding "missing" ordering edges to P_M (for every (a,b) pair that does not have an ordering edge) until it is a (totally) ordered PDG, which uniquely determines a CFG. At first, it might seem that we can simply topologically sort the vertices of P_M in accordance with the existing ordering edges to get a total ordering of the vertices, and use this ordering to fill in "missing" ordering edges. However, this approach is not suitable because the control and flow dependences of P_M may require certain orders. If we are not careful about the addition of the ordering edges, we may end up with a CFG M such that OPDG(M) contains flow or control dependences not present in P_M (or vice versa); in that case, we would not have found a corresponding CFG, and the goal of reconstitution would not be achieved. Instead, we must determine, for all (a,b) pairs in P_M, which of the following holds:

(i) Vertex a postdominates vertex b in all corresponding control-flow graphs.

(ii) Vertex b postdominates vertex a in all corresponding control-flow graphs.

(iii) There is a corresponding control-flow graph in which vertex a postdominates vertex b, and a corresponding control-flow graph in which vertex b postdominates vertex a.

Pairs that fall into categories (i) and (ii) must be ordered $b \longrightarrow_{or} a$ and $a \longrightarrow_{or} b$, respectively; pairs that fall into category (iii) may be ordered nondeterministically as long as the resulting post-domination order over all the pairs is acyclic.

Figure 7.2 gives a high-level view of our reconstitution algorithm. First, P'_M is initialized as a copy of P_M. Second, procedure OrderByControl adds ordering edges to P'_M, as necessitated by control dependences (Section 7.2.4.1). Third, procedure OrderByFlow adds ordering edges to

---

function Reconstitute(P_M: PDG):CFG or FAIL
begin
  $P'_M := P_M$
  if OrderByControl(P'_M) fails **then** return (FAIL) **fi**
  if OrderByFlow(P'_M) fails **then** return (FAIL) **fi**
  M := ConstructCFG(P'_M)
  if OPDG(M) $\neq P'_M$ **then** return (FAIL) **fi**
  return (M)
end

Figure 7.2. Reconstitution of a CFG from the PDG P_M.

P'_M, as necessitated by flow dependences (Section 7.2.4.2). Either of these steps may fail because of contradictory information in P_M, leading to a cycle in the ordering edges. (This can happen if P_M is either infeasible or R-infeasible.) Otherwise, a CFG M consistent with the ordering edges in P'_M is constructed by function ConstructCFG (Section 7.2.4.3). Finally, the isomorphism of OPDG(G) and P'_M is determined. This is necessary because it is possible for P_M to be infeasible yet for the ordering steps (lines [2-3]) to succeed.

7.2.4.1. Procedure OrderByControl

This section focuses on the ordering requirements imposed solely by control dependences. We therefore refer to the control dependence graph (CDG), which is the subgraph of the PDG that includes only control dependence edges. A CDG or (partially) ordered CDG may or may not be feasible, as is the case with a PDG, depending on whether or not the CDG has a corresponding CFG.

DEFINITION (good order). By definition, an order for all (a,b) pairs in a CDG C is a *good order* for C if it is consistent with the postdomination ordering in a corresponding CFG. An infeasible CDG has no good order, while a feasible CDG may have more than one good order.

We introduce several ordering properties that determine when the order of an (a,b) pair is fixed (*i.e.*, the same postdomination relation holds in *all* corresponding CFGs) and show how to determine a good order. These properties are used to define procedure OrderByControl. Unfortunately, these properties are not complete for all feasible CDGs. However, they are complete for the CDGs of normal-form CFGs, which we term R-feasible CDGs. The first two properties are about feasible CDGs: Property *OrderFixed* states that if a certain structural condition holds in a feasible CDG with respect to an (a,b) pair, then b **pd** a (b postdominates a) in all corresponding CFGs; Property *OrderArbitrary* states that if a different condition holds in a feasible CDG, then the postdomination order of the (a,b) pair is not fixed (*i.e.*, there is a corresponding CFG in which a **pd** b and a corresponding CFG in which b **pd** a). Property *Complete* shows that if CDG C is R-feasible then for any (a,b) pair, either property *OrderFixed* or *OrderArbitrary* holds (this is not true for all feasible CDGs). The proofs of these properties can be found in Section 7.3.

DEFINITION (parent(a)-free path). A path in CDG C is *parent(a)-free* iff for all edges $v \longrightarrow_c^L w$ in the path, $v \longrightarrow_c^L a$ is not in C.

DEFINITION (Reach(C,v)). Reach(C,v) denotes the set of vertices reachable from v in C (*i.e.*, $\{ x \mid v \longrightarrow_c^* x \}$).

PROPERTY *OrderFixed*. Consider an (a,b) pair in feasible CDG C. If there is a parent(a)-free path from *ENTRY* to a vertex in Reach(C,b), then b **pd** a in every corresponding CFG of C.

The intuition behind property *OrderFixed* is as follows: if there is a parent(a)-free path from *ENTRY* to x, then there may be some execution of a corresponding CFG in which x executes at least once and a never executes. Suppose that $x \in$ Reach(C,b) and that a **pd** b in some corresponding CFG G of C. By several properties of control dependence, a **pd** b and $x \in$ Reach(C,b) imply that a **pd** x in G. Therefore, any time x is executed, a must eventually execute, which contradicts our claim that x may be executed once while a is never executed. Therefore, b **pd** a in all corresponding CFGs of C if there is a parent(a)-free path from *ENTRY* to a vertex in Reach(C,b).

Example. Consider CDG *C1* in Figure 7.3: in this CDG, there is a parent(a)-free path from the *ENTRY* vertex to b: $ENTRY \longrightarrow_c^T p \longrightarrow_c^T q \longrightarrow_c^F b$. Therefore, b **pd** a in all CFGs corresponding to *C1*. Vertex p is in Reach(*C1*, c) and $ENTRY \longrightarrow_c^T p$ is a parent(q)-free path, so c **pd** q in all CFGs corresponding to *C1*.

DEFINITION (DomReach(C)). Given CDG C, DomReach(C) = $\{ v \mid \forall x \in$ Reach(C,v): v **dom** $x \}$.

Example. DomReach(*C1*) = {*ENTRY*, p, d, q, a, b} and DomReach(*C2*) = {*ENTRY*, q, c, a, b}.

DEFINITION (b **over** a). Given an (a,b) pair in CDG C, b **over** a (by edge $v \longrightarrow_c^L b$) iff there exists $v \longrightarrow_c^L b$ such that (not $v \longrightarrow_c^L a$) and (not b **dom** v).

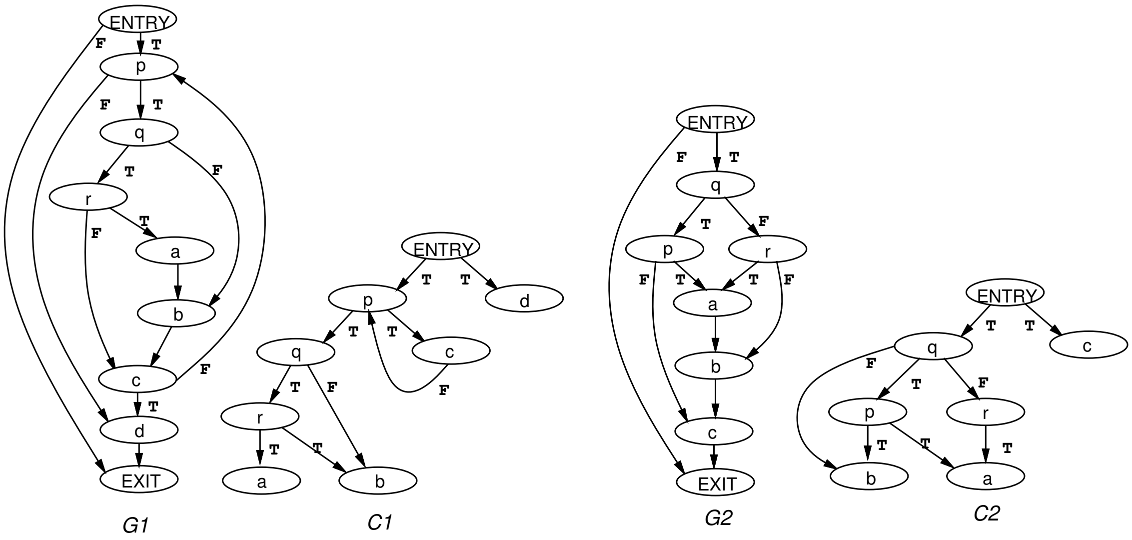


Figure 7.3. Control flow graphs *G1* and *G2* and their control dependence graphs *C1* and *C2*.

Example. In CDG *C1* of Figure 7.3, *b* **over** *a* by edge $q \longrightarrow_c^F b$, and (not *a* **over** *b*).

PROPERTY *OrderArbitrary*. Consider an (a,b) pair in feasible CDG *C*. If not *a* **over** *b*, not *b* **over** *a*, and $a,b \in$ DomReach(*C*), then there is a corresponding CFG in which *b* **pd** *a*, and a corresponding CFG in which *a* **pd** *b*.

Example. Consider CDG *C1* in Figure 7.3. Since *OrderArbitrary* holds for the pair (p,d), there are at least two good orders for *C1*, one in which *d* **pd** *p* and one in which *p* **pd** *d*. In fact, there are exactly two good orders for this CDG, since property *OrderFixed* holds for every other (a,b) pair in *C1*.

In a feasible CDG, it is impossible for both property *OrderFixed* and *OrderArbitrary* to hold for an (a,b) pair. However, there are feasible CDGs for which neither property holds for some (a,b) pair. That is, these properties are not complete for feasible CDGs. However, they are complete for *R*-feasible CDGs:

PROPERTY *Complete*. If CDG C is R-feasible then for any (a,b) pair, either property *OrderFixed* holds (in one direction) or property *OrderArbitrary* holds.

The above three properties give us enough information to determine for which (a,b) pairs the postdomination order is fixed (and in what direction). The following property characterizes how to determine a good order for an R-feasible CDG:

PROPERTY *Permutations*. If CDG C is R-feasible, then an order for the (a,b) pairs of C is a good order iff it (1) respects the fixed pair orderings determined by property *OrderFixed* and (2) orders the (a,b) pairs for which *OrderFixed* does not hold according to an arbitrary total ordering of C's vertices.

Property *Permutations* tells us that we cannot simply use the flip of a coin to determine an order for each (a,b) pair for which property *OrderArbitrary* holds. Cycles in the ordering are not allowed (in particular, because the postdomination relation is acyclic).

Figure 7.4 presents procedure OrderByControl. This procedure makes use of the above properties in conjunction with *regions* of control dependence [17]. Regions partition the vertices of a PDG, grouping vertices with common sets of control dependence parents. Vertices a and b are in the same region iff

$$\{\ (v,L) \mid v \longrightarrow_c^L a \text{ and not } a\ \mathbf{dom}\ v)\ \} = \{\ (v,L) \mid v \longrightarrow_c^L b \text{ and not } b\ \mathbf{dom}\ v)\ \} \neq \emptyset.$$

This is equivalent to: a and b are in the same region iff they are an (a,b) pair, (not a **over** b), and (not b **over** a). In CDG *C1* of Figure 7.3, vertices p and d occupy a region, as do vertices q and c. Every other vertex is in a singleton region.

An (a,b) pair either spans two regions (*i.e.*, a and b are in different regions, R_i and R_j) or vertices a and b are in the same region. If an (a,b) pair spans two regions then it is clearly impossible for property *OrderArbitrary* to hold for this pair. In this case, property *OrderFixed* will determine a postdomination order for the (a,b) pair. Furthermore, for any two (a,b) pairs (call them (c,d) and (e,f)) such that c and e are in R_i and d and f are in R_j, property *OrderFixed* implies that

procedure OrderByControl(P: PDG)
  C: the control dependence subgraph of P
begin
[1] **for** each pair of regions (R_i, R_j) spanned by an (a,b) pair **do**
[2]   apply property *OrderFixed* to (a,b) to order all (a,b) pairs that span R_i and R_j
[3] **od**
[4] **for** each region $R = \{ r_1 \cdots r_n \}$ **do**
[5]   last := $\{ r_i \mid r_i \notin$ DomReach(C) $\}$
[6]   **if** |last| > 1 **then**
[7]   return (FAIL)
[8]   **else if** |last| = 1 **then**
[9]   **let** x be the single vertex in last **in**
[10] $\forall r_i \neq x$, add $r_i \longrightarrow_{or} x$ to P
[11]   **ni**
[12]   **fi**
[13] **od**
end

Figure 7.4. Procedure OrderByControl adds ordering edges to PDG P as determined by control dependences.

d **pd** c iff property *OrderFixed* implies that f **pd** e. Thus, to order all the (a,b) pairs that span the same two regions requires only one application of property *OrderFixed*. Lines [1-3] of procedure OrderByControl reflect this observation.

Suppose that region R contains the vertices $\{r_1, \cdots, r_n\}$. If both r_i and r_j are members of DomReach(C), then property *OrderArbitrary* holds for (r_i, r_j). It is clear that property *OrderFixed* implies that r_i **pd** r_j iff $r_i \notin$ DomReach(C). Conversely, property *OrderFixed* implies that r_j **pd** r_i iff $r_j \notin$ DomReach(C). Therefore, if C is R-feasible then there is at most one r_k such that $r_k \notin$ DomReach(C) (otherwise a cycle would arise in the postdomination order). This implies that if no such r_k exists then property *OrderArbitrary* holds for all pairs in R, and that if such an r_k exists then it must postdominate all other vertices in R and property *OrderArbitrary* holds for all the other pairs in R. We need only examine each vertex in a region once to determine an order for *all* the (a,b) pairs in the region. Lines [4-13] of procedure OrderByControl reflect this observation.

7.2.4.2. Procedure OrderByFlow

As the previous section showed, the order of an (a,b) pair is not fixed by control dependence iff a and b are in the same region and both vertices are in DomReach(C), This section shows how flow dependences may fix the order of such a pair. The ordering requirements imposed by flow dependences in the presence of complex control dependence are quite similar to the structured case, in which the control dependence subgraph is a tree [3].

To understand the operation of procedure OrderByFlow it is necessary to give some more details about regions of control dependence. Suppose that region R contains the vertices $\{r_1 \cdots r_n\}$. Let R' be the subset of vertices in R that are in DomReach(C). Property *OrderArbitrary* holds for every pair of vertices in R' and all the vertices in R' have been ordered before the vertex in $R - R'$ (by procedure OrderByControl). Let $C_i = \text{Reach}(C, r_i)$ and let *AllC* be the union of all the C_i. Figure 7.5 illustrates this structure. Consider an (r_i, r_j) pair. Because r_i and r_j are in the same region there must be a predicate p such that $p \longrightarrow_c^L r_i$, $p \longrightarrow_c^L r_j$, not r_i **dom** p and not r_j **dom** p. These facts imply that C_i and C_j are disjoint sets of vertices. Thus, given a flow edge

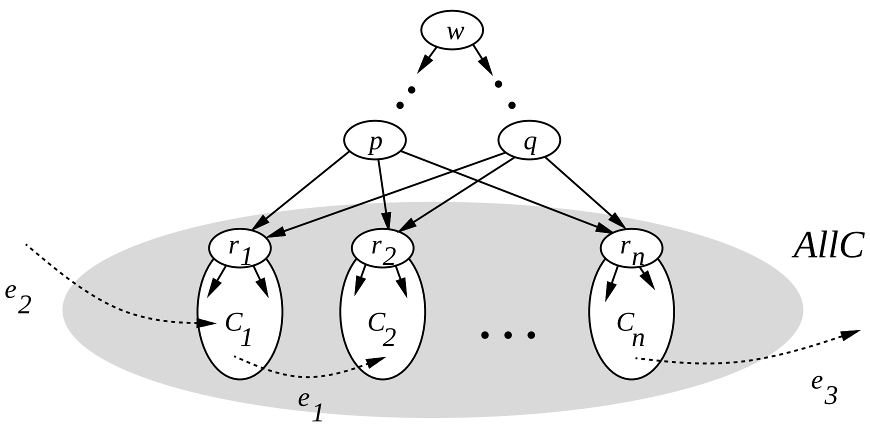


Figure 7.5. Regions of control dependence. Each vertex r_i in the region dominates its reaching set C_i. Each pair of reaching sets (C_i, C_j) is disjoint.

with one or more endpoints inside *AllC*, we can say exactly which C_i contains each endpoint. The only flow dependences that can affect the ordering of the vertices in region R' are the following:

- A flow dependence from a vertex in C_i to a vertex in C_j ($i \neq j$). Edge e_1 in Figure 7.5 is an example of such an edge. In this case there must be an ordering edge between r_i and r_j in P_M (by construction of P_M).
- A flow dependence from a vertex in C_i to a vertex outside of *AllC*, or a flow dependence from a vertex outside of *AllC* to a vertex in C_i. Edges e_2 and e_3 in Figure 7.5 are examples of such edges.
- A loop-carried flow dependence from a vertex in C_i to a vertex in C_j (i and j may be equal in this case) that is carried by a loop entry outside of *AllC*.

No other flow edges in the PDG can affect the ordering of vertices within region R'. Determining the order within each region could be expensive if we had to search each C_i subgraph for the above flow dependences. Procedure OrderByFlow avoids this problem by processing regions in a "bottom-up" manner so that when region R' is considered, all the relevant flow edges will have been projected up to the vertices in the region (*i.e.*, if there is a flow edge from a vertex in C_i to a vertex in C_j then when region R' is considered, this flow edge will have been projected up to be between r_i and r_j). Therefore, ordering the vertices within region R' will only require examining edges with at least one endpoint in R'.

The reason that this bottom-up approach is possible has to do with another property of the vertices within a region: all the vertices in a region have the same immediate dominator in C.[5] Stated another way, all the vertices in a region have the same parent in C's dominator tree. By visiting the vertices in a bottom-up traversal of C's dominator tree, we guarantee that for each vertex r_i in

[5]This follows directly from the observation that the immediate dominator of a vertex x is the least common ancestor in C's dominator tree of all the predecessors of x in C. Since the vertices in a region have the same predecessors in C (except for predecessors that they themselves dominate, which are not a concern), the vertices in a region have the same immediate dominator in C.

region R' (recall that r_i is in DomReach(C)), all the vertices in $C_i - \{r_i\}$ will have been visited before r_i is visited. If there is a flow edge $x \longrightarrow_f y$ such that $x \in C_i$ and $y \notin C_i$ then the source of this edge will have been projected up to r_i (by the processing of vertices in C_i) before r_i is visited.

Figure 7.6 presents procedure OrderByFlow. The procedure makes a copy P_c of the PDG P in which to perform projection of the flow edges. The vertices of the PDG are visited by a bottom-up traversal of the control dependence subgraph's dominator tree (line [2]).[6] If v is the current vertex in the traversal, then R is the region enclosing v. Region R' contains those vertices in R that are in DomReach(C); that is, those vertices for which flow dependences can determine an order (recall that $R - R'$ contains at most one vertex, and that vertex will have been ordered after all other vertices in R by procedure OrderByControl). All vertices in R are marked to ensure that

```
procedure OrderByFlow( P:PDG )
  Pc: a copy of P
  C: the control dependence subgraph of Pc
begin
[1]   unmark all vertices in C
[2]   for each unmarked vertex v in a bottom-up traversal of C's dominator tree do
[3]      let R be the region containing v
[4]           R′ = R − { y | y ∈ R and y ∉  DomReach(C) }
[5]           w = v's parent in C's dominator tree (i.e., w is v's immediate dominator)
[6]      in
[7]          mark all vertices in R
[8]          PreserveExposedUsesAndDefs(Pc ,R′)
[9]          if PreserveSpans(Pc,R′ }) fails then return (FAIL) else TopSort(R′) fi
[10]         ProjectInfo(Pc,w,R)
[11]         add ordering edges between vertices in R′ to PDG P
[12]    ni
[13] od
end
```

Figure 7.6. Procedure OrderByFlow: orderings imposed by flow dependences.

[6]That is, a vertex v in the tree is not visited until all of its descendents in the tree have been visited.

they are not visited a second time in the traversal (line [7]). The procedures PreserveExposedUsesAndDefs and PreserveSpans (described in Sections 7.2.4.2.1-2) add ordering edges between the vertices in region R' to force an ordering of the vertices consistent with the flow dependences (lines [8-9]). If this process introduces a cycle in R', OrderByFlow fails; otherwise, a topological sort of region R' produces an ordering consistent with the region's ordering edges. Procedure ProjectInfo projects flow edge information onto the vertex w that is the immediate dominator of the vertices in R (Section 7.2.4.2.3). Finally, the ordering edges added to region R' in P_c are copied to PDG P (line [11]).

7.2.4.2.1. Procedure PreserveExposedUsesAndDefs

Procedure PreserveExposedUsesAndDefs uses flow edges of graph P_c having only one endpoint inside region R' and loop-carried edges with both endpoints inside R' to identify exposed uses and definitions. It adds ordering edges to R' to ensure that these exposed uses and definitions are ordered correctly with respect to other definitions in R'.

(1) Identify upwards-exposed uses.

> If a vertex $u \in R'$ uses variable x and is the target of a flow edge whose source is outside R', or is the target of a loop-carried flow edge, then vertex u represents an *upwards-exposed* use of x. Mark each such vertex UPWARDS-EXPOSED-USE(x).

(2) Identify downwards-exposed definitions.

> If a vertex $d \in R'$ assigns to variable x and is the source of a flow edge whose target is outside R', or is the source of a loop-carried flow edge then vertex d represents a *downwards-exposed* definition of x. Mark each such vertex DOWNWARDS-EXPOSED-DEF(x).

(3) Preserve exposed uses and definitions.

> For each vertex u marked UPWARDS-EXPOSED-USE(x), add an edge $u \longrightarrow_{or} d$ if vertex $d \in R'$ assigns to variable x and there is no ordering edge $d \longrightarrow_{or} u$. For each vertex d_1 marked DOWNWARDS-EXPOSED-DEF(x), add an edge $d_2 \longrightarrow_{or} d_1$ if vertex $d_2 \in R'$ assigns to variable x and there is no ordering edge $d_1 \longrightarrow_{or} d_2$.

7.2.4.2.2. Procedure PreserveSpans

Consider a definition d_1 in region R' that defines variable x, and the set of uses $u \in R'$ that are flow targets of d_1 and ordered $d_1 \longrightarrow_{or} u$. In general, if there is a definition d_2 to variable x in R' it must either precede d_1 or follow all the uses reached by d_1. Otherwise definition d_2 may inadvertently "capture" one of the uses u.

DEFINITION (Span(d,x)). Let d be a vertex in R' that defines variable x. Span(d,x) = $\{d\} \cup \{u \in R' \mid d \longrightarrow_f^x u \text{ and } d \longrightarrow_{or} u\}$.[7] Span($d$,$x$) is called an x-span and vertex d is its *head*.

Two x-spans are said to be dependent if there is a path of ordering edges from a vertex in one span to a vertex in the other span. The following rule is used to order such dependent spans:

- If there is a path of ordering edges from a vertex in Span(d_1,x) to a vertex in Span(d_2,x) and there is no edge $d_2 \longrightarrow_{or} d_1$, then add an ordering edge from each vertex in Span(d_1,x) – Span(d_2,x) to d_2.

If there is a path of ordering edges from Span(d_1,x) to Span(d_2,x) and no ordering edge $d_2 \longrightarrow_{or} d_1$, then we only add ordering edges from vertices in Span(d_1,x) – Span(d_2,y) to d_2 (rather than adding edges from every vertex in Span(d_1,x) to d_2). The reason for this is that two x-spans may overlap. Figure 7.7 shows the three ways in which two x-spans may overlap. In cases (a) and (b), adding edges from all vertices in Span(d_1,x) to Span(d_2,x) would create a cycle in the ordering edges, making a topological order impossible. Case(c) shows why we insist that there is no edge $d_2 \longrightarrow_{or} d_1$. In this case, it appears that each span should be ordered before the other. Without this condition, the ordering rule would add the edge $d_1 \longrightarrow_{or} d_2$ even though $d_2 \longrightarrow_{or} d_1$ already existed, creating a cycle.

Procedure PreserveSpans operates in our algorithm exactly as in the algorithm for the structured case [3], but uses our updated rule for ordering dependent spans. First, the ordering edges

---

[7] The notation $d \longrightarrow_f^x u$ means that the source of the flow dependence (d) assigns to variable x.

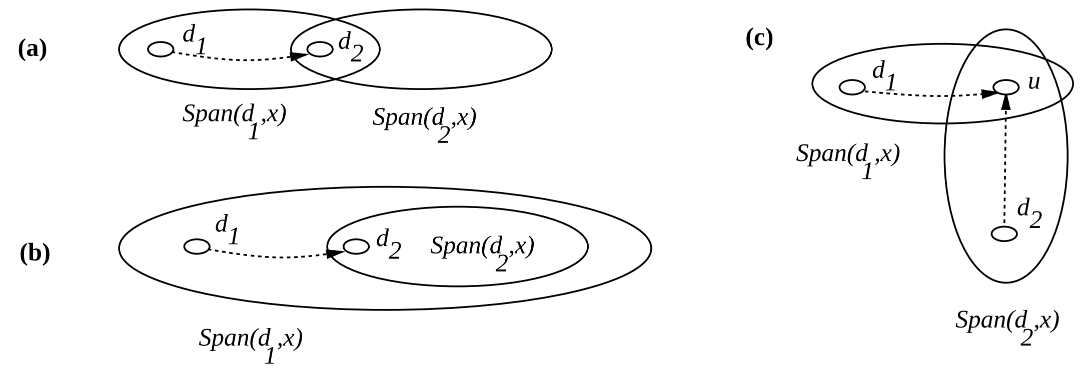


Figure 7.7. The three ways in which *x*-spans may overlap.

in R' are transitively closed (*i.e.*, if $x \longrightarrow_{or} y$ and $y \longrightarrow_{or} z$ then $x \longrightarrow_{or} z$). Second, all dependent spans are ordered. However, after this step there may be independent *x*-spans (*x*-spans with no path of ordering edges between them) that must still be ordered. Although it appears that an arbitrary choice can be made, there are examples in which choosing one way leads to a cycle in the set of ordering edges. As shown by Horwitz, Prins, and Reps, the problem of determining the correct choice in this situation is NP-complete [28]. In practice, a simple backtracking algorithm appears to suffice. Given a pair of independent *x*-spans Span(d_1,x) and Span(d_2,x) an ordering edge is added (either $d_1 \longrightarrow_{or} d_2$ or $d_2 \longrightarrow_{or} d_1$) to make them dependent, the ordering edges in R' are transitively closed, and the spans are ordered by applying the rule for ordering dependent spans. If a cycle is introduced in the ordering edges during this process, PreserveSpans backtracks to the most recent choice point and tries the other choice. If all choices lead to a cycle then PreserveSpans fails. Ball, Horwitz, and Reps proved the correctness of this approach [3].

7.2.4.2.3. Procedure ProjectInfo

Procedure ProjectInfo projects flow edges with one or more endpoints inside region R to the vertex w in the control dependence subgraph C that dominates the vertices in R. This ensures that if $w \in$ DomReach(C) when w is considered in its enclosing region, it represents all uses and definitions that occur in Reach(C, w). The procedure takes the following steps:

(1) Let V be the set of variables that are defined by vertices in R. Label w as defining every variable in V.

(2) Replace all flow edges $a \longrightarrow_f^x b$ such that w does not dominate a in C and $b \in R$ with an edge $a \longrightarrow_f^x w$. Label w as representing a use of variable x. Replace all flow edges $a \longrightarrow_f^x b$ such that $a \in R$ and w does not dominate b in C with an edge $w \longrightarrow_f^x b$.

(3) Consider each loop-carried flow edge $d \longrightarrow_f^x u$ such that d and u are both in R. If the edge $d \longrightarrow_f^x u$ is labelled with loop entry w then remove the edge; otherwise replace the edge (labelled with loop entry z) with a loop-carried flow edge $w \longrightarrow_f^x w$ (labelled with loop entry z) and label w as representing a use of variable x.

7.2.4.3. Function ConstructCFG: constructing the CFG

This section shows how to construct a corresponding CFG from P'_M. For each predicate p in P'_M, the L control dependence successors of p are totally ordered by the ordering edges of P'_M. Let the list $CLIST(p, L) = (v_1, ..., v_m)$ be the L control dependence successors of p ordered so that for all i, $1 \le i < m$, $v_i \longrightarrow_{or} v_{i+1}$. The following rules describe how CFG postdomination information can be propagated in a top-down fashion over P'_M and used to determine the edge set of the desired CFG M (a CFG that corresponds to P'_M):

- The immediate postdominator of the *ENTRY* vertex in any CFG is the *EXIT* vertex.
- If vertex w immediately follows v in CLIST(p,L), then $w = \text{ipd}(G,v)$.
- If vertex v occurs last in CLIST(q,L) and $w = \text{ipd}(G,q)$ then $w = \text{ipd}(G,v)$.

These rules provide the basis for the function ConstructCFG of Figure 7.8, which first builds the CLISTs and then calls the procedure DFS to produce the edge set of the CFG via a depth-first search of the control dependence subgraph of P'_M. The first parameter to DFS (v) is the current

```
global
  CLIST : (vertex, LABEL) → list of vertices;
  Edges : set of edges


function ConstructCFG(
    P′M: fully ordered PDG
                              ): CFG
begin
    Edges := ∅
    unmark all vertices in P′M
    DFS(ENTRY,EXIT)
    return( (vertices(P′M) ∪ {EXIT}, Edges) )
end
```

```
procedure DFS( v,w : vertices );
 begin
   if not Marked(v) then
       Marked(v) := TRUE;
       if v is a predicate then
          for LBL := false to true do
             if CLIST(v,LBL) = () then
                Edges := Edges ∪ { v →^LBL w }
             else
                curr := head(CLIST(v,LBL));
                rest := tail(CLIST(v,LBL));
                Edges := Edges ∪ { v →^LBL curr }
                while rest ≠ () do
                   DFS(curr,head(rest))
                   curr := head(rest); rest := tail(rest)
                od
                /* curr is now the last vertex */
                DFS(curr,w)
             fi
          od
       else  /* v is a fall-through vertex */
          Edges := Edges ∪ { v → w }
       fi
   fi
 end
```

Figure 7.8. Function ConstructCFG uses a depth-first search over the control dependences of P'_M to generate the edges for the CFG M.

vertex of the depth-first search and the second parameter (w) is always the immediate postdominator of v, which is updated in accordance with the above rules. The edges of the control-flow graph are determined as follows:

- The CFG successor of a vertex that is not a predicate is its immediate postdominator.
- The CFG L-successor of a predicate p such that CLIST(p,L) $\neq$ () is the first vertex in CLIST(p,L).
- The CFG L-successor of a predicate p such that CLIST(p,L) = () is p's immediate postdominator.

If P'_M is feasible then function ConstructCFG will return a *corresponding* CFG for C. Otherwise, ConstructCFG will return a CFG that does not correspond to C. Section 7.3 proves the correctness of function ConstructCFG.

Example. We now show how function ConstructCFG produces a CFG from an ordered CDG. Consider the *R*-feasible CDG in Figure 7.9(a). Thin edges represent control dependences while thick edges represent ordering edges. There is an ordering edge between each (a,b) pair (for readability, we have omitted transitive edges). Property *OrderFixed* only holds for two (a,b) pairs;

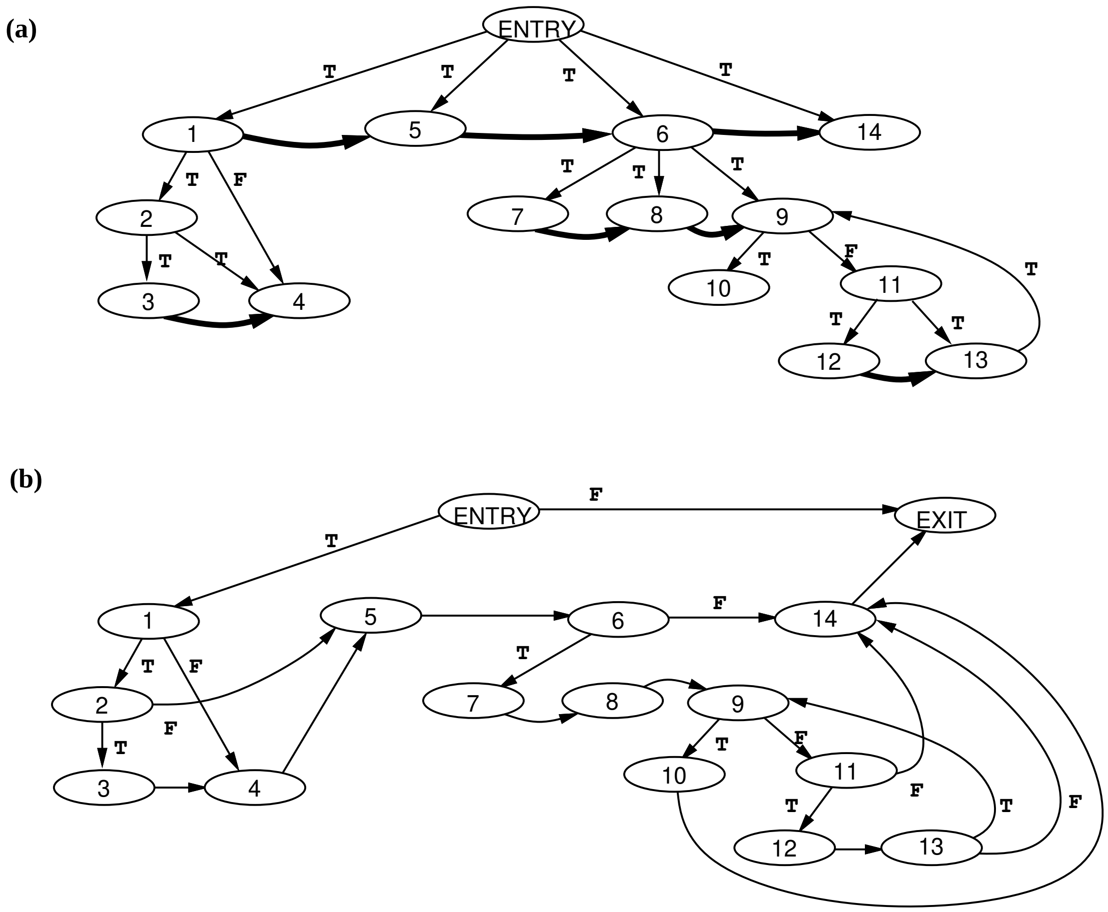


Figure 7.9. (a) An *R*-feasible CDG with ordering edges; (b) the corresponding CFG that would be constructed by function ConstructCFG.

it determines that 4 **pd** 3 and 13 **pd** 12. Property *OrderArbitrary* holds for every other (a,b) pair. Thus, we are free to give any total order to the *T*-successors of *ENTRY* and to the *T*-successors of vertex 6 (note that $(10,11)$ is not an (a,b) pair because 10 and 11 do not share a common *L*-parent). We have chosen to order these vertices as shown in the figure but emphasize that there are other good orders for this CDG.

The graph in Figure 7.9(b) is the corresponding CFG that results from applying function ConstructCFG to the CDG and the good order. We illustrate how a portion of the CFG is built. Figure 7.10 shows some of the CLISTs and a portion of the execution of function DFS (see Figure 7.8), which creates the edge set of the CFG. Recall that the second vertex in the call to DFS is always the immediate postdominator of the first. DFS is initially invoked (line [1]) with the call DFS(*ENTRY*,*EXIT*). Since CLIST(*ENTRY*,*F*) is empty, the edge $ENTRY \rightarrow^{F} EXIT$ is made (line [2]). The edge $ENTRY^{T}\!\rightarrow 1$ is added because 1 is the first vertex in CLIST(*ENTRY*,*T*) (line [3]). Since 5 follows 1 in CLIST(*ENTRY*,*T*), 5 is passed down as the immediate postdominator of 1 in the call DFS(1,5) (line [4]). The edge $1 \rightarrow^{F} 4$ is added because 4 is the first vertex in CLIST(1,*F*).

Some CLISTs				Execution of DFS
CLIST(*ENTRY*,*F*)	=	()	[1]	DFS(*ENTRY*,*EXIT*)
CLIST(*ENTRY*,*T*)	=	(1,5,6,14)	[2]	add edge $ENTRY \rightarrow^{F} EXIT$
CLIST(1,*F*)	=	(4)	[3]	add edge $ENTRY \rightarrow^{T} 1$
CLIST(1,*T*)	=	(2)	[4]	DFS(1,5)
CLIST(2,*F*)	=	()	[5]	add edge $1 \rightarrow^{F} 4$
CLIST(2,*T*)	=	(3,4)	[6]	DFS(4,5)
			[7]	add edge $4 \rightarrow 5$
			[8]	add edge $1 \rightarrow^{T} 2$
			[9]	DFS(2,5)
			[10]	add edge $2 \rightarrow^{F} 5$
			[11]	add edge $2 \rightarrow^{T} 3$
			[12]	DFS(3,4)
			[13]	add edge $3 \rightarrow 4$
			[14]	DFS(4,5)
			[15]	DFS(5,6)
				....

Figure 7.10. Some CLISTs and a portion of the execution of function DFS.

Since 4 is also the last vertex in CLIST(1,*F*), 5 is passed down as the immediate postdominator of 4 in the call DFS(4,5) (line [6]). Because 4 is a fall-through vertex, the edge 4→5 is added (line [7]). At this point, we return to process CLIST(1,*T*) in the active call DFS(1,5).

7.3. PROOFS

This section contains the proofs of the results stated in this chapter. Section 7.3.1 reviews some basic results about control dependence. Section 7.3.2 proves that $FPP(G,v) = FPP(H,w)$ iff OPDG(*G*)/*v* = OPDG(*H*)/*w* and proves the correctness of function ConstructCFG. Section 7.3.3 proves the correctness of the control dependence ordering properties *OrderFixed*, *OrderArbitrary*, *Complete*, and *Permutations*.

7.3.1. Some Basic Results About Control Dependence

LEMMA(7.1). If CFG *G* has vertices *v*, *w*, and *z* such that *z* **pd** *v* and $v \longrightarrow_c w$ is in CDG(*G*), then there is a non-empty *z*-free path from *v* to *w* in *G*.

PROOF. By contradiction. Since $v \longrightarrow_c w$ there must be a non-empty path from *v* to *w*. Suppose that every such path contains *z*. The facts that not *w* **pd** *v* and *z* **pd** *v* imply that not *w* **pd** *z*. Because *z* is on every path from *v* to *w* and not *w* **pd** *z*, *w* cannot postdominate the *L*-branch of *v*, as required by $v \longrightarrow_c^L w$. Contradiction. □

LEMMA(7.2). If CFG *G* has vertices *v*, *w*, and *z* such that *z* **pd** *v* and $v \longrightarrow_c w$ in CDG(*G*), then *z* **pd** *w*.

PROOF. By lemma (7.1), there must be a non-empty *z*-free path from *v* to *w*. If there were a *z*-free path from *w* to *EXIT* then *z* could not postdominate *v*. Therefore, *z* **pd** *w*. □

LEMMA(7.3). If CFG *G* has vertices *a*, *b*, and *x* such that *b* **pd** *a* and $a \longrightarrow_c^+ x$ in CDG(*G*), then *b* **pd** *x*.

PROOF. Let $a \longrightarrow_c x_1 \longrightarrow_c \cdots \longrightarrow_c x_n$ be a control dependence path from *a* to *x* where $x_n = x$. By lemma (7.2), *b* **pd** *a* and $a \longrightarrow_c x_1$ implies *b* **pd** x_1. Given *b* **pd** x_i and $x_i \longrightarrow_c x_{i+1}$, $1 \le i < n$, lemma (7.2) implies that *b* **pd** x_{i+1}. Therefore, *b* **pd** *x*. □

LEMMA(7.4). $v \longrightarrow_c^+ w$ in CDG(G) iff there is a non-empty path from v to w in G that does not contain any postdominators of v.

PROOF.

($\Rightarrow$) Let $v \longrightarrow_c x_1 \longrightarrow_c \cdots \longrightarrow_c x_n$ be a control dependence path from v to w where $x_n = w$. By lemma (7.3), any postdominator of v must postdominate all x_i. Since $v \longrightarrow_c x_1$, there must be a non-empty path from v to x_1. By lemma (7.1), one of these paths contains no postdominators of v. By similar reasoning, for all i ($1 \le i < n$), since $x_i \longrightarrow_c x_{i+1}$ and any postdominator of v postdominates x_i, there must be a non-empty, path from x_i to x_{i+1} that contains no postdominators of v. Therefore, there must be a non-empty path from v to w that does not contain any postdominators of v.

($\Leftarrow$) Let *PTH* be a path from v to w that contains no postdominators of v. The proof is by induction on the number of predicates in *PTH* (not including w). If there is one predicate in *PTH*, it must be v (if v were a fall-through vertex then *PTH* contains a postdominator of v). Since every other vertex in *PTH* is a fall-through vertex, w must postdominate a branch of v. Since not w **pd** v, it follows that $v \longrightarrow_c w$. The induction hypothesis is that if there are fewer than N predicates in *PTH*, then $v \longrightarrow_c^+ w$. The induction step follows: since *PTH* is contains no postdominators of v, the induction hypothesis implies that all the predicates in *PTH* (except v and w) are transitively control dependent on v. Let p be the last predicate in *PTH* such that not w **pd** p (such a p must exist since not w **pd** v). It is clear that w postdominates every fall-through and predicate vertex between p and w in *PTH*. Therefore, w postdominates a branch of p and $p \longrightarrow_c w$ exists. Since $v \longrightarrow_c^+ p$, it follows that $v \longrightarrow_c^+ w$. □

LEMMA(7.5). If a **pd** v in CFG G (where a is not the *EXIT* vertex), then every path from *ENTRY* to v in CDG(G) must include an edge $p \longrightarrow_c^L x$ such that $p \longrightarrow_c^L a$ and a **pd** x in G (*i.e.*, there is no path from *ENTRY* to v that is parent(a)-free).

PROOF. By induction on the length of the path *PTH* from *ENTRY* to v in CDG(G).

Base Case: length of *PTH* = 1. In this case, *PTH* = $ENTRY \longrightarrow_c^T v$ and $p = ENTRY$ and $x = v$. Since $a \neq EXIT$, it follows that not a **pd** *ENTRY*. Furthermore, since a **pd** v and v postdominates the T-branch of $ENTRY$, it follows that a postdominates the T-branch of $ENTRY$. Therefore, $ENTRY \longrightarrow_c^T a$ exists. Since $x = v$ and a **pd** v, it follows that a **pd** x.

Induction Hypothesis: If a **pd** v then for all PTH of length $< N$, there is an edge $p \longrightarrow_c^L x$ in PTH such that $p \longrightarrow_c^L a$ and a **pd** x.

Induction Step: PTH of length $= N$. Suppose the control dependence parent of v in PTH ($p \longrightarrow_c^L v$) is not postdominated by a. Since a **pd** v it follows that $p \longrightarrow_c^L a$. Instead, suppose a **pd** p. $p \longrightarrow_c^L a$ obviously cannot exist. Let PTH_p be the prefix of PTH up to predicate p. The length of this path is less than N. Therefore, by the induction hypothesis, since a **pd** p, there must be a $q \longrightarrow_c^{L'} x$ in PTH_p such that $q \longrightarrow_c^{L'} a$ and a **pd** x. □.

7.3.2. PDG Isomorphism and Function ConstructCFG

The results on ordered PDG isomorphism and the correctness of function ConstructCFG rely on the following four lemmas that relate the structure of a CFG G to the structure of its ordered CDG C. In the following lemmas, let the list $CLIST(p,L) = (v_1,...,v_m)$ be the L-children of p in CDG C ordered so that for all i, $1 \le i < m$, $v_i \longrightarrow_{or} v_{i+1}$ is in C.

LEMMA (7.6). Let G be a CFG with ordered CDG C. If w immediately follows v in CLIST(p,L), then $w = \text{ipd}(G,v)$.

PROOF. By contradiction. Suppose x ($\neq w$) is the immediate postdominator of v in G. Since x **impd** v and w **pd** v in G (since w occurs after v in CLIST(p,L)), w **pd** x **pd** v in G. Since $p \longrightarrow_c^L w$ and $p \longrightarrow_c^L v$, and w **pd** x **pd** v, it follows that $p \longrightarrow_c^L x$ is in C. Therefore, x must occur between v and w in CLIST(p,L). Contradiction. □

LEMMA (7.7). Let G be a CFG with ordered CDG C. If vertex v occurs last in CLIST(q,L) and $r = \text{ipd}(G,q)$, then $r = \text{ipd}(G,v)$.

PROOF. Assume that $w \neq r$ is the immediate postdominator of v in G. Since r **impd** q in G and v is directly control dependent on q, lemma (7.2) implies that r **pd** v in G. Since w **pd** v and r **pd** v in G, either w **pd** r or r **pd** w must hold in G. The former implies that w is not the immediate postdominator of v in G. Therefore, r **pd** w **pd** v. Since v postdominates the L-branch of q and w

pd v, but not w **pd** q (since r **pd** w), $q \longrightarrow_c^L w$ must be in C. However, since w **pd** v, w must follow v in CLIST(q,L), contradicting an initial assumption. □

LEMMA (7.8). Let G be a CFG with ordered CDG C. The immediate postdominator of every vertex in G is determined solely by the structure of C.

PROOF. The proof is by induction on the length of an acyclic path in C from *ENTRY* to a vertex v. The base case is a path length of 0, in which case $v = ENTRY$. In any CFG, the immediate postdominator of *ENTRY* is always *EXIT*. Suppose that $r = \mathrm{ipd}(G,q)$ and that there is an acyclic path of length N from *ENTRY* to q. By the Induction Hypothesis, r can be determined solely by the structure of C. We show that the immediate postdominator of each vertex v in CLIST(q,L) is determined solely by the structure of C, completing the induction step. If v is not the last vertex in CLIST(q,L) then lemma (7.6) implies that $w = \mathrm{ipd}(G,v)$, where w immediate follows v in CLIST(q,L). If v is the last vertex in CLIST(q,L) then lemma (7.7) implies that $r = \mathrm{ipd}(G,v)$. □

LEMMA (7.9). Let G be a CFG with ordered CDG C. The edge set of G is determined solely by the structure of C.

PROOF. In any CFG, the control-flow successor of a fall-through vertex is its immediate postdominator. By lemma (7.8), the immediate postdominator of every vertex in G is determined solely by the structure of C. Therefore, so is the successor of every fall-through vertex. We now show that the *true* and *false* control-flow successors of each predicate p in G are determined solely by the structure of C. There are two cases to consider:

(1) p has no L-successors in C. In this case, ipd(G,p) is the L-successor of p in G. Suppose that x ($x \neq \mathrm{ipd}(G,p)$) is the L-successor of p in G. This implies that $p \longrightarrow_c^L x$ is in C, which contradicts the fact that p has no L-successors in C.

(2) p has L-successors in C. In this case, the first vertex in CLIST(p,L) is the L-successor of p in G. Let x be the L-successor of p in G. Vertex x cannot postdominate p, otherwise there would be no L-successors of p in C. This implies that $p \longrightarrow_c^L x$. Since x is the L-successor of p in G, it must be the first vertex in CLIST(p,L) (because every other vertex that is L

control dependent on p must postdominate x). □

THEOREM. Given CFGs G and H containing vertices v and w, respectively. Let $G' = FPP(G,v)$ and $H' = FPP(H,w)$. $G' = H'$ iff OPDG(G)/v = OPDG(H,w)

PROOF. The proof has two major parts:

(1) Given CFG G, if $G' = FPP(G,v)$ then OPDG(G') = OPDG(G)/v.

(2) Given CFGs I and J, $I = J$ iff OPDG(I) = OPDG(J).

The theorem follows directly from these points: Since $G' = FPP(G,v)$ and $H' = FPP(H,w)$, point (1) implies that OPDG(G') = OPDG(G)/v and OPDG(H') = OPDG(H)/w. This fact and point (2) implies that $G' = H'$ iff OPDG(G)/v = OPDG(H)/w.

We first prove point (1). By definition, G' has the same vertex set as OPDG(G)/v (except for the *EXIT* vertex). Since G' is a path-projection of G, it follows that for any two vertices (a,b) in G', a **pd** b in G' iff a **pd** b in G and a postdominates the L-branch of b in G' iff a postdominates the L-branch of b in G. Therefore, for any two vertices (a,b) in G', $a \longrightarrow_c^L b$ is in OPDG(G') iff $a \longrightarrow_c^L b$ is in OPDG(G)/v. Since the postdomination order of vertices in G' and G is the same, it is clear that $a \longrightarrow_{or} b$ is in OPDG(G') iff $a \longrightarrow_{or} b$ is in OPDG(G)/v. To complete the proof of point (1) we must show that $a \longrightarrow_f b$ is in OPDG(G') iff $a \longrightarrow_f b$ is in OPDG(G)/v. Consider a path in G that induces a flow dependence $a \longrightarrow_f b$ in OPDG(G)/v. The projection of this path in G' clearly induces the flow dependence $a \longrightarrow_f b$ in OPDG(G'). Consider a path P' in G' that induces a flow dependence $a \longrightarrow_f b$ in OPDG(G'). Let P be a generating path in G for P' that begins with a and ends with b. Assume that vertex a assigns to variable x. Since G' is a flow/path-projection of G, no vertex in P before b, except a, can assign to x (otherwise by point (2) of the definition of flow/path-projection, that vertex would also be in $V(G')$, and by point (2) of the definition of path-projection, it would occur in P' before b). This implies that path P induces the flow dependence $a \longrightarrow_f b$.

We now prove point (2). It is clear that if CFGs I and J are isomorphic that OPDG(I) = OPDG(J). The other direction follows directly from lemma (7.9). By this lemma, the edge set of I is completely determined by the control dependences and ordering edges of OPDG(I), and likewise for J. Therefore, if OPDG(I) = OPDG(J), it follows that $I = J$. □

THEOREM. Let G be a CFG with ordered CDG C. Given CDG C, function ConstructCFG will build CFG G.

PROOF. This follows directly from lemma (7.8) and (7.9). By lemma (7.8), it is clear that for any call to DFS, vertex w is the immediate postdominator of vertex v in CFG G. Since function ConstructCFG correctly determines the immediate postdominators, lemma (7.9) implies that it correctly determines the edge set of G. □

7.3.3. Proofs of CDG Ordering Properties

Section 7.3.3.1 proves property *OrderFixed* and Section 7.3.3.2 proves property *OrderArbitrary*. Section 7.3.3.2 shows that properties *OrderFixed* and *OrderArbitrary* are complete for acyclic feasible CDGs. Section 7.3.3.3 proves that R-feasible CDGs are reducible. Section 7.3.3.4 shows that removing the backedges from an R-feasible CDG yields a feasible CDG. Using the results of the previous three sections, Section 7.3.3.5 proves property *Complete*: properties *OrderFixed* and *OrderArbitrary* are complete for (possibly cyclic) R-feasible CDGs. Finally, Section 7.3.3.6 proves property *Permutations*.

7.3.3.1. Property *OrderFixed*

PROPERTY *OrderFixed*. Consider an (a,b) pair in feasible CDG C. If there is a parent(a)-free path *PTH* from *ENTRY* to a vertex in Reach(C,b), then b **pd** a in every corresponding CFG of C.

PROOF. By contradiction. Assume all conditions, except that a **pd** b in some G (since a and b share a common parent, either b **pd** a or a **pd** b must hold). Let v be the vertex in Reach(C,b) to which there is a parent(a)-free path from *ENTRY*. By lemma (7.3), $b \longrightarrow_c^* v$ and a **pd** b imply that a **pd** v. Lemma (7.5) implies that *PTH* is not parent(a)-free, a contradiction. □

7.3.3.2. Property *OrderArbitrary*

PROPERTY *OrderArbitrary*. Consider an (a,b) pair in feasible CDG C. If not a **over** b, not b **over** a, and $a,b \in$ DomReach(C), then there is a corresponding CFG in which b **pd** a, and a corresponding CFG in which a **pd** b.

The correctness of property *OrderArbitrary* follows immediately from the following lemma, which is used in the proof of property *Permutations*.

LEMMA (7.10). Consider an (a,b) pair in feasible CDG C and a corresponding CFG in which b **pd** x **pd** a (or b **impd** a). If not a **over** b, not b **over** a, and $a,b \in$ DomReach(C), then there is a corresponding CFG in which a **pd** x **pd** b (a **impd** b).

PROOF. Consider the special case where a and b are statement vertices. Since a and b have no outgoing edges in C, neither dominates any vertex besides itself in C. This fact together with (not a **over** b) and (not b **over** a) implies that the sets $\{ (v,L) \mid v \longrightarrow_c^L a \}$ and $\{ (v,L) \mid v \longrightarrow_c^L b \}$ are equal. Thus, if G is a corresponding CFG in which b **pd** x **pd** a (or b **impd** a), then simply renaming a to b and vice-versa yields a corresponding CFG G' in which a **pd** x **pd** b (a **impd** b).

To extend this proof to the case where a and b are predicates requires the notion of a control-flow hammock. A hammock is a single-entry, single-exit subgraph of the CFG (it is useful to think of a hammock as a "super vertex"). When the conditions of property *OrderArbitrary* hold, it is possible to swap the hammocks with entries a and b in a corresponding CFG in which b **pd** a to yield a corresponding CFG in which a **pd** b. The remainder of this section proves this in greater detail.

DEFINITION (hammock). A *hammock* H is an induced subgraph of CFG G with entry vertex v in H and exit vertex v' not in H such that:

1. All edges from $(G-H)$ to H go to v.
2. All edges from H to $(G-H)$ go to v'.

LEMMA(7.11). Consider a vertex v in CDG C with corresponding CFG G. Let H = Reach(C,v). Vertex v dominates all vertices in H in C iff H is a hammock in G where v is the entry and ipd(G,v) is the exit.

PROOF.

($\Leftarrow$) By lemma (7.4), every vertex in H must be transitively control dependent on v (because no vertex in H can postdominate v). Since H is a hammock, all edges from $G-H$ to H go to v. These facts imply for any vertex $w \in H$ and $w \neq v$, w cannot be directly control dependent on a vertex outside of H. Therefore, v dominates all vertices in H in C.

($\Rightarrow$) We show that the two conditions for H (Reach(C,v)) to be a hammock must hold:

(1) Suppose that there is an edge $x \rightarrow y$ in CFG G where $x \notin$ Reach(C,v), $y \in$ Reach(C,v), and $v \neq y$. If not y **pd** x in G, then $x \longrightarrow_c y$, which contradicts the assumption that v dominates Reach(C,v). Therefore, y **pd** x in G. By lemma (7.5), along any path from *ENTRY* to x in C there must be an edge $p \longrightarrow_c^L z$ in the path such that $p \longrightarrow_c^L y$. If $p \notin$ Reach(C,v) then v cannot dominate Reach(C,v). If $p \in$ Reach(C,v), then since $p \longrightarrow_c^+ x$, it follows that $x \in$ Reach(C,v). Contradiction.

(2) Suppose that there is an edge $x \rightarrow y$ in CFG G where $x \in V(H)$, $y \in V(G-H)$, and $y \neq$ ipd(G,v). Since $v \longrightarrow_c^+ x$, lemma (7.4) implies that there is a from v to x in G containing no postdominators of v. If not y **pd** v then there is a path from v to y containing no postdominators of v and lemma (7.4) implies that $v \longrightarrow_c^+ y$. This implies that $y \in$ Reach(C,v), a contradiction of $y \in V(G-H)$. Therefore, y **pd** v. However, since $v \longrightarrow_c^+ x$, there must be a path from v to x in G that does not contain a postdominator of v. Therefore, there is a path from v to y in G in which y is the first postdominator of v in the path. This implies that $y =$ ipd(G,v). Contradiction. $\square$

We are now in a position to prove lemma (7.10) in its full generality. We assume that the following hold for the given (a,b) pair: not a **over** b, not b **over** a, and $a,b \in$ DomReach(C). We assume, without loss of generality, that C has a corresponding CFG G in which b **pd** x **pd** a (or b **impd** a). We must show that C also has a corresponding CFG G' in which a **pd** x **pd** b (a **impd** b).

Since both a and b are in DomReach(C), lemma (7.11) implies that G must contain a hammock H_a = Reach(C,a) with entry a and exit ipd(G,a), and a hammock H_b = Reach(C,b) with entry b and exit ipd(G,b). Hammocks are either disjoint or nested. Since b **pd** a in CFG G and no vertex in H_a can postdominate a, H_a and H_b must be disjoint.

The CFG G' contains the same vertex set as G. The edge set of CFG G' is defined as follows:

$$
\begin{aligned}
& \{\, v \to^L w \mid v \to^L w \text{ is in } G \text{ and } [\, (v,w \notin H_a \cup H_b) \text{ or } (v,w \in H_a) \text{ or } (v,w \in H_b)\,]\,\} \\
\cup\quad & \{\, v \to^L b \mid v \to^L a \text{ is in } G \text{ and } v \notin H_a \,\} \\
\cup\quad & \{\, v \to^L a \mid v \to^L b \text{ is in } G \text{ and } v \notin H_b \,\} \\
\cup\quad & \{\, v \to^L \mathrm{ipd}(G,b) \mid v \to^L \mathrm{ipd}(G,a) \text{ is in } G \text{ and } v \in H_a \,\} \\
\cup\quad & \{\, v \to^L \mathrm{ipd}(G,a) \mid v \to^L \mathrm{ipd}(G,b) \text{ is in } G \text{ and } v \in H_b \,\}
\end{aligned}
$$

The relationship between CFGs G and G' is depicted in Figure 7.11. It is obvious that a **pd** x **pd** b (or a **impd** b) in CFG G' and that H_a and H_b are hammocks in CFG G'. To show that G' corresponds to C we must show that CDG(G') is identical to C. This follows from the four observations below (the last three have symmetrical counterparts replacing a with b and H_a with H_b):

(1) Since hammocks are single-entry and single-exit subgraphs, swapping hammocks H_a and H_b cannot affect any control dependence between vertices v and w that are both outside of $(H_a \cup H_b)$.

(2) Since swapping the hammocks does not change any paths strictly inside H_a and does not change any postdomination ordering between vertices strictly inside H_a, there is no effect on control dependence between vertices v and w that are both inside H_a.

(3) Since (not a **over** b) and (not b **over** a) in CDG C, swapping the hammocks cannot affect the control dependence predecessors of a that lie outside of H_a.

(4) Since H_a is a hammock in both G and G', lemma (7.11) implies that there can be no control dependence in either C or CDG(G') from a vertex inside H_a to a vertex outside of H_a, or from a vertex outside of H_a to a vertex inside of H_a other than a. □

7.3.3.3. Completeness for acyclic feasible CDGs

In this section we show that either property *OrderFixed* or property *OrderArbitrary* holds for every (a,b) pair in an acyclic feasible CDG (it is clearly impossible for *OrderFixed* to hold in both directions in a feasible CDG). The proof relies on the following definition and lemma:

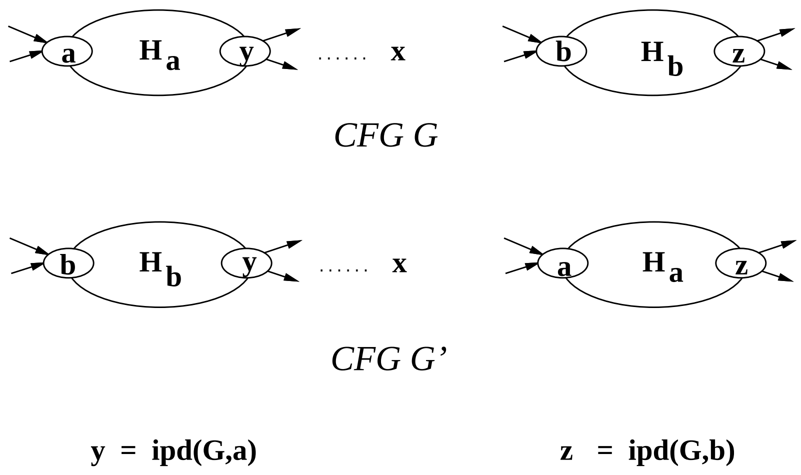


Figure 7.11.

LEMMA (7.12). There is a cycle in CFG G iff there is a cycle in CDG(G).

PROOF. Straightforward, from the definition of control dependence. □

LEMMA (7.13). Consider an (a,b) pair in CDG C of CFG G. If there is a vertex x such that $x \in$ Reach(C,a) and $x \in$ Reach(C,b), then there is a cycle in C.

PROOF. Without loss of generality, assume that b **pd** a in G. This implies that $x \neq b$ (if $x = b$ then $a \longrightarrow_c^+ b$, which is not possible since b **pd** a). Since $x \in$ Reach(C,a) and b **pd** a, lemma (7.3) implies that b **pd** x. There must be a path in G from b to x (because $b \longrightarrow_c^+ x$) and a path in G from x to b (because b **pd** x); therefore, b and x participate in a cycle in G. By lemma (7.12), there must be a cycle in CDG C. □

DEFINITION (parent(a,b)-free path). A path in a CDG is parent(a,b)-free iff for each edge $x \longrightarrow_c^L y$ in the path, not ($x \longrightarrow_c^L a$ and $x \longrightarrow_c^L b$).

LEMMA (7.14). Consider an (a,b) pair in CDG C. If there is no parent(a)-free path (from *ENTRY*) to a vertex in Reach(C,b), and no parent(b)-free path to a vertex in Reach(C,a), then for any z in Reach(C,a) $\cup$ Reach(C,b), there is no parent(a,b)-free path from *ENTRY* to z.

PROOF. By contradiction. Assume the antecedent but that there is a z in Reach(C,a) $\cup$ Reach(C,b) such that there is a parent(a,b)-free path *PTH* from *ENTRY* to z. Consider the parent(a)-free and parent(b)-free prefix (possibly empty) of *PTH*. Let $p \longrightarrow_c^L x$ be the edge in *PTH* immediately after the prefix. Either $p \longrightarrow_c^L a$ or $p \longrightarrow_c^L b$ (but not both) exists. This implies that there is parent(b)-free path to a or a parent(a)-free path to b (consisting of the prefix followed by the edge $p \longrightarrow_c^L a$ or $p \longrightarrow_c^L b$). Contradiction. □

LEMMA (7.15). Properties *OrderFixed* and *OrderArbitrary* are complete for acyclic feasible CDGs.

PROOF. By contradiction. We suppose that there is an (a,b) pair in an acyclic feasible CDG C for which neither property holds and yield the contradiction that C must contain a cycle. If property *OrderFixed* does not hold in either direction for an (a,b) pair, then there is no parent(a)-free path (from *ENTRY*) to a vertex in Reach(C,b), nor is there a parent(b)-free path to a vertex in Reach(C,a). Lemma (7.14) implies that for any z in Reach(C,a) $\cup$ Reach(C,b), there is no parent(a,b)-free path from *ENTRY* to z. Suppose that property *OrderArbitrary* does not hold, either because (1) $a \notin$ DomReach(C) or (2) a **over** b:

(1) Suppose that $a \notin$ DomReach(C) because there is some $v \notin$ Reach(C,a) and $w \in$ Reach(C,a) such that $v \longrightarrow_c w$ and $w \neq a$. Since there can be no parent(a,b)-free path from *ENTRY* to w, on every path from *ENTRY* to w that ends with $v \longrightarrow_c w$ there must be an edge $x \longrightarrow_c^L y$ such that $x \longrightarrow_c^L a$ and $x \longrightarrow_c^L b$. If $x = v$ then $y = w$ and $y \neq a$ (because $w \neq a$). If $x \neq v$ then $y \neq a$ (otherwise $v \in$ Reach(C,a)). In either case, vertices y and a are distinct, share a common control parent (x) and reach a common vertex (w), so lemma (7.13) implies that there must be a cycle in C. Contradiction.

(2) Suppose that a **over** b by edge $v \longrightarrow_c^L a$. Since there can be no parent(a,b)-free path from *ENTRY* to a, on every path from *ENTRY* to v there must be an edge $x \longrightarrow_c^L y$ such that $x \longrightarrow_c^L a$ and $x \longrightarrow_c^L b$. Since a **over** b by $v \longrightarrow_c^L a$, $v \neq x$, which implies that $y \neq a$. Vertices y and a are distinct, share a common control parent (x) and reach a common vertex (a), yielding the same contradiction as in (1). □

7.3.3.4. *R*-feasible CDGs are reducible

This section shows that *R*-feasible CDGs are reducible. There are many equivalent definitions of reducibility. A graph G (rooted at *ENTRY*) is reducible iff any of the following is true:

(1) For each back edge $v \rightarrow w$ of G (as identified by a depth-first search of G starting at the *ENTRY* vertex) w **dom** v.

(2) Every depth-first search of G identifies the same set of edges as backedges.

(3) For any cycle in G, there is a vertex v in the cycle that dominates all other vertices in the cycle.

The following lemma relates domination in the CFG to domination in the CDG:

LEMMA (7.16). If v dominates w in reducible CFG G and $v \longrightarrow_c^* w$ CDG(G), then v dominates w in CDG(G).

PROOF. Suppose that v dominates w in reducible CFG G and that $v \longrightarrow_c^* w$ CDG(G). Let H be the set of vertices that are dominated by v and postdominated by ipd(G,v) in G. Suppose there is a v-free path in CDG(G) from *ENTRY* to w. Since $ENTRY \notin H$ and $w \in H$, this path must include an edge $z \longrightarrow_c z'$ such that $z \notin H$ and $z' \in H$. We will show that no vertex in H other than v can be directly control dependent on a vertex z outside H; thus, this path cannot exist and v dominates w in CDG(G). There are two cases to consider, depending on whether or not there is a v-free control-flow path from z to a vertex in H:

(1) Every control-flow path from z to a vertex u in H contains v. If $u = v$ then u can be directly control dependent on z. However, if $u \neq v$, then u cannot be directly control dependent on z, as we now argue. Since every control-flow path from z to u must pass through v and no vertex in H can postdominate v, it is impossible for u to postdominate the *T*-branch (or *F*-branch) of z. Thus, u could not be control dependent on z.

(2) There is a v-free control-flow path *PTH* from z to a vertex in H. Without loss of generality assume that the last vertex (u) in *PTH* is in H and that all other vertices in *PTH* are not in H. Let $t \rightarrow u$ be the last edge in *PTH*. We show that u cannot dominate t in G and that there is a depth-first search of G that identifies $t \rightarrow u$ as a backedge. This implies that G is irreducible,

yielding a contradiction.

It should be clear that no vertex in H (except v) can dominate a vertex outside of H. All that remains to be shown is that there is a cycle-free path from *ENTRY* to t that includes u (which implies that some depth-first search would identify $t \rightarrow u$ as a backedge). There is a cycle-free path PTH_0 from *ENTRY* to v such that no vertex in PTH_0 (except v) is in H. There is a cycle-free path PTH_1 from v to ipd(G,v) that includes u such that every vertex in PTH_1 (except ipd(G,v)) is in H. There is a cycle-free path PTH_2 from ipd(G,v) to t such that no vertex in PTH_2 is in H. The only way a cycle could arise in the concatenation of the three paths is if PTH_2 and PTH_0 share a vertex x ($x \neq v$, since no vertex in PTH_2 is in H). Concatenating the prefix of PTH_0 up to and including x, the suffix of prefix of PTH_2 starting at x, and the edge $t \rightarrow u$, yields a v-free path from *ENTRY* to u, contradicting u's membership in H. Therefore ($PTH_0 \parallel PTH_1 \parallel PTH_2$) must be cycle-free. □

LEMMA (7.17). If CFG G is a normal-form CFG (*i.e.*, CDG(G) is R-feasible) then C = CDG(G) is reducible.

PROOF. We show that for any cycle $Cycle_C$ in C, there is a vertex in $Cycle_C$ that dominates all other vertices in $Cycle_C$ in C. In particular, we show that there is a loop-entry w such that all the vertices in $Cycle_C$ belong to nat-loop(w) and that w is in $Cycle_C$. Since w dominates all vertices in nat-loop(w) in G (because G is reducible), lemma (7.16) implies that w dominates all vertices in $Cycle_C$ in C.

Let $Cycle_C$ be represented by the path of control dependences $v_1 \longrightarrow_c v_2 \cdots \longrightarrow_c v_n$, where $v_1 = v_n$. For each control dependence $v_i \longrightarrow_c^L v_{i+1}$ in $Cycle_C$, there is path P_i in G from v_i to v_{i+1} such that:

(1) P_i begins with the L-branch of v_i.

(2) v_{i+1} postdominates every vertex in P_i except v_i (otherwise v_{i+1} could not postdominate the L-branch of v_i or v_{i+1} **pd** v_i, either of which would imply that $v_i \longrightarrow_c^L v_{i+1}$ could not exist in C) and itself.

Linked together, these paths form a cycle $Cycle_G$ in CFG G. There must be some backedge $v \rightarrow w$ in G in $Cycle_G$ such that every vertex in $Cycle_G$ is in nat-loop(w). We need only show that there is a v_i such that $w = v_i$ in order to complete the proof. Since every vertex in $Cycle_G$ is in nat-loop(w) and every natural loop in G is a while loop (as G is a normal-form CFG), it follows that no vertex in $Cycle_G$ can postdominate w. Consider a subpath P_i of $Cycle_G$ that contains w. By point (2) above, vertex v_{i+1} postdominates every vertex in P_i except v_i and itself. Therefore, either $w = v_i$ or $w = v_{i+1}$. □

7.3.3.5. The feasibility of backedge-free *R*-feasible CDGs

Given a reducible CDG C, its backedge-free counterpart (BF(C)) is uniquely defined. While BF(C) is an acyclic CDG, it is not necessarily a feasible CDG. However, we show that for any R-feasible CDG C, BF(C) is feasible.

LEMMA(7.18). If CDG C is R-feasible, then BF(C) is feasible.

PROOF. Given a CFG G in the restricted class, we show that BF(CDG(G)) is feasible. The proof is by induction on the number of backedges in G.

Base Case: If G has no backedges then, by lemma (7.12), CDG(G) has no backedges either. This implies that BF(CDG(G)) = CDG(G).

Induction Hypothesis: If G has fewer than N backedges ($N > 0$) then BF(CDG(G)) is feasible.

Induction Step: If CFG G has N backedges ($N > 0$) then BF(CDG(G)) is feasible. The proof consists of three steps:

(1) Find an *outermost* loop in CFG G headed by loop entry w and pick a backedge $v \rightarrow w$. Construct CFG G' from G as follows: G' = (vertices(G),edges(G)−$\{v \rightarrow w\} \cup \{v \rightarrow \text{ipd}(G,w)\}$).

(2) Show that G' is in the restricted class and contains fewer than N backedges. This implies that CDG(G') is reducible, so BF(CDG(G')) is uniquely defined.

(3) Show that BF(CDG(G)) = BF(CDG(G')).

Since G' is in the restricted class and contains fewer than N backedges, the Induction Hypothesis implies that BF(CDG(G')) is feasible. Since BF(CDG(G)) = BF(CDG(G')), BF(CDG(G)) is feasible.

We now turn to proving (2) and (3). The proofs of these steps rely on the following properties relating postdomination and domination in G and G'. In the remainder of this section, z denotes the vertex ipd(G,w). The proofs make use of the following two observations: any $v \rightarrow w$-free path in G is also in G'; any $v \rightarrow z$-free path in G' is also in G.

(a) $\forall x,\ \forall y$: not y **pd** x in $G \Rightarrow$ not y **pd** x in G'.

If there is a y-free path from x to *EXIT* in G that is $v \rightarrow w$-free, then the same path is clearly in G'. If the only y-free path from x to *EXIT* in G contains $v \rightarrow w$, then there must be a y-free and $v \rightarrow w$-free path from x to v and from w to *EXIT* (and from z to *EXIT*, since z **pd** w) in G. The same paths are in G'. Since v is connected to z by $v \rightarrow z$ in G', it follows that not y **pd** x in G'.

(b) $\forall y,\ y \neq w$: not y **pd** x in $G' \Rightarrow$ not y **pd** x in G.

If there is a y-free path from x to *EXIT* in G' that is $v \rightarrow z$-free, then the same path is in G. If the only y-free path *PTH* from x to *EXIT* in G' contains $v \rightarrow z$, then there must be a y-free and $v \rightarrow z$-free path from x to v and from z to *EXIT* in G'. Thus, the same paths are in G. If there is a y-free path from w to z in G, then not y **pd** x in G. If y is on every path from w to z in G, then since z = ipd(G,w), it follows that either $y = w$ or $y = z$. The first case contradicts the assumption that $y \neq w$. The second case implies that *PTH* is not y-free.

(c) $\forall x,\ x \notin$ nat-loop(w) in G : not w **pd** x in $G' \Rightarrow$ not w **pd** x in G.

If there is a w-free path from x to *EXIT* in G' that is $v \rightarrow z$-free, then the same path is in G. If the only w-free path *PTH* from x to *EXIT* in G' contains $v \rightarrow z$, then there must be a w-free and $v \rightarrow z$-free path from x to v in G', and thus in G. However, since $x \notin$ nat-loop(w) and $v \in$ nat-loop(w), any path from x to v in G must contain w.

(d) $\forall x,\ \forall y$: not y **dom** x in $G \Rightarrow$ not y **dom** x in G'.

If there is a y-free path from *ENTRY* to x in G that is $v \rightarrow w$-free, then the same path is in G'. It is not possible that the only y-free path from *ENTRY* to x in G contains $v \rightarrow w$, since w **dom** v in G.

(e) $\forall x,\ \forall y$: not y **dom** x in G' $\Rightarrow$ not y **dom** x in G.

If there is a y-free path from *ENTRY* to x in G' that is $v \rightarrow z$-free, then the same path is in G. If the only y-free path *PTH* from *ENTRY* to x in G' contains $v \rightarrow z$, then there is a y-free path *PTH′* (a prefix of *PTH*) from *ENTRY* to v and a y-free path from z to x in G. Since w **dom** v in G, w occurs in *PTH′* and thus in *PTH*. If there is a y-free path from w to z in G, then there is a y-free path from *ENTRY* to x in G. If y is on every path from w to z in G, then since z = ipd(G,w), either $y = z$ or $y = w$. In the former case, *PTH* cannot be y-free. In the latter case, since $y = w$ and w is in *PTH*, *PTH* cannot be y-free.

- G' **is reducible**.

Since z = ipd(G,w) and G is in the restricted class, z cannot be in nat-loop(w). This and the fact that nat-loop(w) is an outermost loop imply that w and z cannot participate in a loop in G. Because z = ipd(G,w), there can be no path from z to w in G. It follows that there can be no path from z to a member of nat-loop(w) in G. Thus, the edge $v \rightarrow z$ cannot participate in any loop in G'.

A graph is irreducible iff it contains a cycle such that there is no vertex in the cycle that dominates all other vertices in the cycle. Suppose that such a situation occurs in G'. Since $v \rightarrow z$ cannot participate in this loop, the same loop exists in G. Furthermore, (e) implies that if not y **dom** x in G' then not y **dom** x in G. This implies that G is irreducible.

- G' **contains fewer than** N **backedges**

Since G and G' are reducible and y **dom** x in G iff y **dom** x in G' (by (d) and (e)), it follows that any backedge in G (excluding $v \rightarrow w$) is a backedge in G', and that any backedge in G' (with the exception of $v \rightarrow z$) is a backedge in G. We show that $v \rightarrow z$ cannot be a backedge in G' (not z **dom** v in G'), which implies that G' contains $N-1$ backedges. As shown above, z and v cannot participate in a loop in G. Since there is a path from v to z in G (as $v \rightarrow w$ and z = ipd(G,w)), it follows that not z **dom** v in G. Thus, (d) implies that not z **dom** v in G'.

- G' **is in the restricted class**.

We prove that for each loop entry d in G', no vertex in nat-loop(d) postdominates d. As shown above, every backedge in G' is also a backedge in G. Since $v \to z$ cannot participate in any loop, for each backedge $c \to d$ in G', nat-loop($c \to d$) in G' is equal to nat-loop($c \to d$) in G. Consider any vertex f in nat-loop($c \to d$) in G'. Vertex f is also in nat-loop($c \to d$) in G. Since G is in the restricted class it follows that not f **pd** d. By (a), not f **pd** d in G'. Thus, G' is in the restricted class.

- **BF(CDG(**G**)) = BF(CDG(**G'**))**.

We prove that BF(CDG(G)) = BF(CDG(G')) by showing the following relationships between CDG(G) and CDG(G'):

(i) $\forall x,\ \forall y, y \neq w : x \longrightarrow_c^L y$ in CDG(G) $\Leftrightarrow$ $x \longrightarrow_c^L y$ in CDG(G').

From (a) and (b) it follows that if $y \neq w$ then not y **pd** x in G iff not y **pd** x in G', and that y postdominates the L-branch of x in G iff y postdominates the L-branch of x in G'. Therefore, if $y \neq w$, then $x \longrightarrow_c^L y$ in CDG(G) iff $x \longrightarrow_c^L y$ in CDG(G').

(ii) $\forall x, x \notin$ nat-loop(w) in G : $x \longrightarrow_c^L w$ in CDG(G) $\Leftrightarrow$ $x \longrightarrow_c^L w$ in CDG(G').

From (a) and (c) it follows that if $x \notin$ nat-loop(w), then not w **pd** x in G iff not w **pd** x in G', and that w postdominates the L-branch of x in G iff w postdominates the L-branch of x in G'.

(iii) $\forall x,\ x \in$ nat-loop(w) in G : w **dom** x in CDG(G) and CDG(G'). This implies that if $x \longrightarrow_c^L w$ is in CDG(G) or CDG(G'), then it is a backedge of that CDG.

Since $x \in$ nat-loop(w) in G, w **dom** x in G. By (e), w **dom** x in G'.

Since G is reducible and no member of nat-loop(w) can postdominate w in G, there is a path PTH from w to x in G containing no postdominators of w (without loss of generality, we can assume that PTH is $v \to w$ free, since inclusion of $v \to w$ would imply that PTH contains a cycle). Since PTH is $v \to w$-free in G, PTH exists in G'. Since no vertex in PTH postdominates w in G, (a) implies that no vertex in PTH postdominates w in G'. Thus, lemma (7.4) implies that $w \longrightarrow_c^+ x$ in CDG(G) and in CDG(G').

Since w **dom** x in G and G', and $w \longrightarrow_c^+ x$ in CDG(G) and CDG(G'), lemma (7.16) implies that w **dom** x in CDG(G) and in CDG(G').

(iv) $\forall x,\ \forall y : y$ **dom** x in CDG(G) $\Leftrightarrow$ y **dom** x in CDG(G').

Follows from (i), (ii), and (iii). We prove ($\Leftarrow$). The proof of ($\Rightarrow$) is symmetrical (switch CDG(G) and CDG(G')). Suppose that not y **dom** x in CDG(G). Let *PTH* be a y-free path from *ENTRY* to x in CDG(G). Note that we can always pick *PTH* such that for each edge $a \longrightarrow_c b$ in *PTH*, not b **dom** a in CDG(G). *PTH* cannot contain an edge of the form $a \longrightarrow_c w$, where $a \in$ nat-loop(w) in G, because (iii) implies that w **dom** a in CDG(G). Thus, (i) and (ii) imply that *PTH* is in CDG(G') and not y **dom** x in CDG(G').

The above four points imply that the only edges on which CDG(G) and CDG(G') can differ are backedges (edges for which the target dominates the source in the CDG). Thus, BF(CDG(G)) = BF(CDG(G')). □

7.3.3.6. Completeness for *R*-feasible CDGs

In this section we show that properties *OrderFixed* and *OrderArbitrary* are complete for *R*-feasible CDGs. The following lemmas characterize some important relationships between *R*-feasible CDG C and BF(C), C's backedge-free counterpart.

LEMMA(7.19). If CDG C is *R*-feasible, then

(1) b **over** a in BF(C) iff b **over** a in C, and

(2) $b \notin$ DomReach(BF(C)) $\Rightarrow$ $b \notin$ DomReach(C)

PROOF.

(1) ($\Leftarrow$) If b **over** a by $v \longrightarrow_c^L b$ in C, then $v \longrightarrow_c^L b$, not b **dom** v, and not $v \longrightarrow_c^L a$. Since C is reducible and not b **dom** v, it follows that $v \longrightarrow_c^L b$ cannot be a backedge. Thus, b **over** a by $v \longrightarrow_c^L b$ in BF(C).

($\Rightarrow$) Suppose b **over** a in BF(C) by $v \longrightarrow_c^L b$ (*i.e.*, $v \longrightarrow_c a$ is not in BF(C) and not b **dom** v in BF(C)). Since C is reducible, it follows that y **dom** x in C iff y **dom** x in BF(C). Therefore, not b **dom** v in C. Suppose that $v \longrightarrow_c^L a$ is in C. Since $v \longrightarrow_c^L a$ is a backedge it follows that a **dom** v in C, and thus in BF(C). This means that $a \longrightarrow_c^+ b$ in BF(C). Lemma (7.13) implies that there is a loop in BF(C). Contradiction.

(2) If $b \notin$ DomReach(BF(C)), then there must be a vertex x ($x \neq b$) in Reach(BF(C),b) such that there is a b-free path from *ENTRY* to x in BF(C). The same path must obviously exist in C.

LEMMA(7.20). Given an (a,b) pair in R-feasible CDG C. If $b \in$ DomReach(BF(C)) and $b \notin$ DomReach(C), then there is a parent(a)-free path from *ENTRY* to a member of Reach(C,b).

PROOF. Since C is reducible, b **dom** x in BF(C) iff b **dom** x in C. If no vertex in Reach(BF(C),b) is the source of a backedge in C, then $b \in$ DomReach(C). If for every vertex v in Reach(BF(C),b) that is the source of a backedge $v \longrightarrow_c w$ in C, w is in Reach(BF(C),b), then $b \in$ DomReach(C).

The only case left is where there is a vertex v in Reach(BF(C),b) that is the source of a backedge $v \longrightarrow_c w$ in C, and $w \notin$ Reach(BF(C),b). Since C is reducible, w must dominate b in this case. We show that w **dom** a, which implies that there must be a parent(a)-free path to w (a member of Reach(C,b)).

Let p be the common parent of vertices a and b. Since w **dom** b and $p \longrightarrow_c b$, either $w = p$ or w **dom** p. It follows that $w \longrightarrow_c^+ a$ and that $b \longrightarrow_c^+ a$. Since a and b have a common parent, either b **pd** a or a **pd** b in a corresponding CFG. However, by lemma (7.4), if a **pd** b in a corresponding CFG then $b \longrightarrow_c^+ a$ cannot exist. Therefore, b **pd** a in all corresponding CFGs. Suppose that not w **dom** a by some path *PTH*. Since b **pd** a in all corresponding CFGs, lemma (7.5) implies that there is some $x \longrightarrow_c^L y$ in *PTH* such that $x \longrightarrow_c^L b$. This implies that there is an w-free path from *ENTRY* to b consisting of the prefix of *PTH* up to x, followed by the edge $x \longrightarrow_c^L b$. This contradicts the fact that w **dom** b. □

LEMMA(7.21). If C is R-feasible and there is a parent(a)-free path from *ENTRY* to a vertex in Reach(C,b) in BF(C), then the same path is parent(a)-free in C.

PROOF. By contradiction. If the path is not parent(a)-free in C, then it must be because of a backedge ($v \longrightarrow_c a$). Since C is reducible, it follows that a **dom** v. This implies that a is in the parent(a)-free path in BF(C), which is clearly a contradiction as a parent(a)-free path cannot contain a. □

PROPERTY *Complete*. Properties *OrderFixed* and *OrderArbitrary* are complete for R-feasible CDGs.

PROOF. To show that properties *OrderFixed* and *OrderArbitrary* are complete for a given R-feasible CDG C, we consider which property holds in BF(C). Since BF(C) is feasible and acyclic, one of the properties must hold for a given (a,b) pair.

(A) Assume that property *OrderFixed* holds (in some direction) in BF(C) for (a,b). Lemma (7.21) implies that property *OrderFixed* holds in the same direction in C.

(B) Assume that property *OrderArbitrary* holds in BF(C) for (a,b). If property *OrderArbitrary* holds in C, then we are done. If it does not hold, then lemma (7.19) part (1) implies that either a or b is not a member of DomReach(C). Lemma (7.20) implies that property *OrderFixed* must hold in C. □

7.3.3.7. **Property** *Permutations*

The proof of property *Permutations* relies on the following lemma.

LEMMA (7.23). Given feasible CDG C. If property *OrderArbitrary* holds for $(a,\ b)$ and b **pd** x **pd** a in a corresponding CFG of C, then (1) there is a vertex p such that $p \longrightarrow_c^L a$, $p \longrightarrow_c^L x$, and $p \longrightarrow_c^L b$, and (2) property *OrderArbitrary* holds for (a,x) and (x,b).

PROOF. There must be a vertex p such that $p \longrightarrow_c^L a$ and $p \longrightarrow_c^L b$. Since b **pd** x **pd** a, it follows that $p \longrightarrow_c^L x$. By lemma (7.10), there must be a corresponding CFG G' in which a **pd** x **pd** b, which implies (2). □

PROPERTY *Permutations*. If CDG C is R-feasible, then an order for the (a,b) pairs of C is a good order iff it (1) respects the fixed pair orderings determined by property *OrderFixed* and (2) orders the (a,b) pairs for which property *OrderArbitrary* holds according to an arbitrary total ordering of C's vertices.

PROOF.

$(\Rightarrow)$ If *Order* is a good order, then *Order* must be acyclic and respect the fixed orderings of property *OrderFixed*. Since *Order* is acyclic it must be possible to find a total ordering of C's vertices that respects the pair orderings of those (a,b) pairs for which property *OrderArbitrary* holds.

$(\Leftarrow)$ Since property *OrderArbitrary* cannot hold for an (a,b) pair that spans regions and regions partition the vertex set of C, we only need to argue that within any region the (a,b) pairs for which property *OrderArbitrary* holds can be ordered according to an arbitrary order of the vertices in that region. Within any region $R = \{ x_1,...,x_n \}$ of an R-feasible CDG C, at most one x_k cannot be a member of DomReach(C). Such an x_k must postdominate all other vertices in the region. Property *OrderArbitrary* holds for every pair of vertices from the set $S = (R - \{ x_k \mid x_k \notin \text{DomReach}(C)\})$.

Consider a corresponding CFG of C, and let x_n be the vertex from S such that x_n postdominates all other vertices in S and let x_1 be the vertex from S such that all other vertices in S postdominate x_1. This chain contains all the vertices in S and no others (if there was an $x_k \notin S$ such that x_n **pd** x_k **pd** x_1 then lemma (7.23) implies that property *OrderArbitrary* holds for (x_n,x_k) and (x_k,x_1) which implies that $x_k \in S$). Since property *OrderArbitrary* holds for every pair in the chain, lemma (7.10) implies that there is a corresponding CFG for every possible postdomination ordering of vertices in S. $\square$

7.4. RELATED WORK

7.4.1. Related Work on Differencing

There are many techniques for determining the difference between two programs. Some of these techniques are based solely on program text and others take program semantics into account. The Unix™ utility *diff* computes the textual difference between two files [35]. An extension of this tool, called *spiff*, is a differential comparator that uses lexical parsing of files to provide comparisons at the level of tokens [51]. For example, *spiff* can be told to ignore whitespace and comments when examining files or to ignore the differences between floating point numbers up to some threshold (this can be useful for comparing the output of floating point calculations in the presence of roundoff errors). Yang developed a tool called *cdiff* that computes the syntactic difference of two C programs [72]. This tool goes a step beyond *spiff* by representing programs as abstract syntax trees rather than a stream of tokens and comparing the abstract syntax trees. None of the above tools take program semantics into account and so they may not report differences in sections of code that actually exhibit different behavior.

There are a number of works on semantics-based differencing. Yang and Reps showed that the isomorphism of subgraphs induced by backwards-closure in the program dependence graph (a slightly different form of graph from the one defined in Chapter 5) can be used to compare the behavior of components in different (structured) programs [62]. Horwitz and Reps gave an efficient procedure for determining the isomorphism of these PDG subgraphs [33]. We have extended the former result to show how ordered program dependence graphs can be used to compare the behavior of components in programs with arbitrary control-flow. Furthermore, we have shown that isomorphism of these ordered graphs and subgraphs can be decided efficiently, as done by Horwitz and Reps for the structured case.

Yang, Horwitz, and Reps defined a different algorithm for determining components with equivalent behavior across programs that uses a representation called the *program representation graph* [73], which has some similarities to static single assignment form [15]. Although this algorithm is able to find more cases where components have equivalent behavior, it is still limited to programs with structured control-flow. Binkley describes how to perform semantics-based differencing on multi-procedure programs with applications to regression testing [9].

7.4.2. Related Work on Integration

The first semantics-based integration algorithm (the HPR algorithm) was developed by Horwitz, Prins and Reps [30]. Yang and Reps proved that when this integration algorithm succeeds it will produce a program meeting the semantic criteria of integration [62]. The HPR algorithm is restricted to programs with structured control-flow. Our algorithm has the same basic steps as the HPR algorithm: identification of vertices with different and equivalent behavior (*affected* and *preserved* behavior in the terminology of the HPR algorithm) and construction of merged dependence graph P_M; Type I interference test; reconstitution of a CFG (program) from P_M. There are three major differences between the two algorithms.

- As the HPR algorithm deals only with structured programs, the problem of reconstitution is greatly simplified (because the control dependence subgraph of the merged dependence graph is a tree). In fact, when the control dependence subgraph is a tree there are no ordering constraints imposed by control dependences. The correctness of the simplified reconstitution algorithm was proved correct by Ball, Horwitz, and Reps [3]. In contrast, our algorithm must deal with ordering constraints imposed by control dependences as well as flow dependences. This was the major difficulty in extending the integration algorithm to handle more complex control-flow.
- The result of the HPR algorithm is a program while the result of our integration algorithm is a CFG. Because the HPR algorithm deals only with structured programs, it is straightforward to map the merged dependence graph back to a program (in fact, the control dependence subtree closely resembles the abstract syntax tree, except that children of a vertex in the abstract syntax tree are totally ordered). However, in the presence of complex control-flow there are multiple programs that have the same control-flow and dependence graph representation. We have chosen to produce a CFG rather than a program, sidestepping the problem of which program to generate from the output CFG M. This has the advantage that

the integration algorithm is independent of the source language and its syntax. Ultimately, one would like integration to produce a program (and perhaps one that resembles the source programs as much as possible). One could use a structuring algorithm such as Baker's to produce a program from the CFG [2], but there is no guarantee that this program will resemble the source programs. Another approach is to carry along syntactic information from the source programs through the integration process, and use this to guide the creation of an appropriate program from the output CFG *M*.

- The program dependence graph representation used by the HPR allows the algorithm to identify vertices that have equivalent behavior in the presence of reordering. To repeat an example from Chapter 6, the HPR algorithm would identify vertices $c := a + b$ in the following two programs as having equivalent behavior:

a := 1	b := 2
b := 2	a := 1
c := a+b	c := a+b

  On the other hand, our algorithm identifies these vertices as having (potentially) inequivalent behavior. While our algorithm allows for a greater class of control-flow, there are cases (such as the one above) where our algorithm will identify vertices as having (potentially) inequivalent behavior when the HPR algorithm will identify the vertices as having equivalent behavior. This will allow the HPR algorithm to succeed in some cases where our algorithm will fail.

Binkley extended integration to handle programs with multiple procedures [8] and Yang developed a new integration algorithm that is able to accommodate semantics-preserving transformations on programs [73]. Both of these algorithms assume structured control-flow within a procedure or program.

7.4.3. Related Work on Reconstitution

Horwitz, Prins, and Reps were the first to formalize that program dependence graphs are an "adequate" program representation [29]. They proved that PDGs distinguish between nonequivalent *structured* programs. That is, any two structured programs with inequivalent behavior have non-isomorphic PDGs. This result is important to HPR integration because it implies that all programs that could be created from the merged PDG by reconstitution have equivalent behavior (at each corresponding component). While we have not explicitly stated an adequacy result for ordered PDGs, a similar adequacy result follows directly from our results. If G and H are CFGs and v and w are corresponding vertices from G and H, respectively, with inequivalent behavior, then OPDG(G)/v cannot be isomorphic to OPDG(H)/v. When it succeeds, our reconstitution algorithm guarantees that for each vertex v in the CFG M constructed from the merged dependence graph P_M, OPDG(M)/v = P_M/v. Therefore, all the CFGs that could be created from P_M by reconstitution have equivalent behavior (at each corresponding vertex).

Previous work on PDG reconstitution has been done by Ferrante and Mace [16], Ferrante, Mace and Simons [18], and Simons, Alpern and Ferrante [65]. In these works, the PDG is viewed as a representation of a parallel program and the goal is to translate this program to a sequential program. In order to do this, ordering must be introduced between the control dependence successors of a predicate vertex, which is the same problem we have addressed. All three works consider control dependence that may not be tree-shaped. The first two works were incomplete and contained some errors that were corrected in [65], which takes the most formal approach to PDG reconstitution. We discuss the similarities and differences between that work and our work on PDG reconstitution. Both works handle essentially the same class of PDGs, as we discuss later. However, the formulation of our ordering properties are different due to a different representation of control dependence. Furthermore, we present a much more complete description of how flow dependences order vertices within a region. Simons et al. do not address the issue of upwards-exposed uses, downwards-exposed definitions, or dependent and independent x-spans, as we do.

Two aspects of the work of [65] that appear different but that are actually only superficially so, have to do with the class of control dependence graphs that can be handled. Both the algorithm presented in this chapter and the algorithms of [65] handle restricted classes of feasible CDGs.

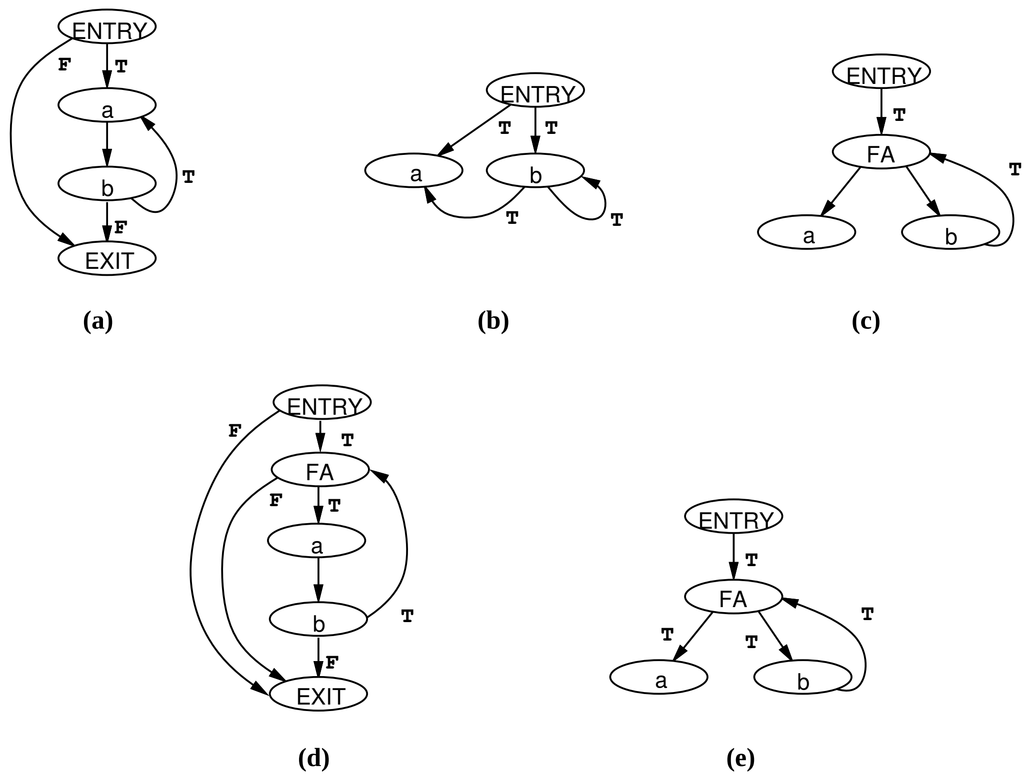


Figure 7.12. (a) A CFG that is not in normal form (vertex *b* postdominates vertex *a* which is a loop entry). (b) The CDG of this CFG. (c) The CDG, as transformed by Simons et al. (d) Translation of the CFG in (a) into normal form. (e) The CDG of the CFG in (d).

In [65] these restrictions are defined directly in terms of properties of the CDGs. In contrast, we define the restrictions *indirectly* by saying that we handle only those CDGs that correspond to normal-form CFGs. Simons et al. restrict the CDGs to be reducible and have certain structure that essentially makes them isomorphic to the CDGs of normal-form CFGs. We illustrate this with an example. Consider the CFG in Figure 7.12(a). Vertex *a* is a loop entry but vertex *b*, which is in the loop headed by *a*, postdominates *a* (*i.e.*, the loop is not a while loop). Case (b) shows the control dependence graph of this CFG. This CDG does not meet the Simon's requirements. The CDG in case (c) shows the form in which they need the CDG, which requires the insertion of the vertex *FA* (called a *forall* vertex, which groups vertices *a* and *b* into the same region). On the other hand, consider the CFG in case (d). If we consider the vertex *FA* as a predicate that always evaluates to true, then this CFG has the same set of executable paths as the CFG in case (b). However, the loop in this CFG is a while loop. Case (e) shows the control dependence graph of this CFG, which is clearly isomorphic to Simon’s CDG in case (c) (except that in their representation the outgoing edges of a forall vertex have no labels). In this way, we arrive at the same class of CDGs.

Another aspect of the question of what CDGs are handled has to do with *region* vertices (also called *forall* vertices), which are assumed in [65] to have been added explicitly to the CDG (region vertices are added to a CDG to gather all vertices with the same set of control conditions together). Their ordering properties are stated in terms of control dependence paths and the forall vertices, whereas our properties are stated only in terms of control dependence paths. However, our algorithm (procedure OrderByControl in Figure 7.4) uses regions to reduce the number of ordering queries that must be made.

Chapter 8

CONCLUSIONS AND FUTURE WORK

This thesis has shown how the control-flow and control dependence information in programs can be used in a variety of software tools to collect information about program executions (profiling, tracing, and event counting) and to help with the debugging and maintenance of programs (slicing, differencing, and integration).

Chapters 3 and 4 showed how the control-flow graph can be used to instrument programs for a variety of analyses that provide information about the execution behavior of a program over a single run: *profiling* measures the execution frequency of basic blocks or control-flow edges, *tracing* records the sequence of basic blocks traversed in an execution, and *event counting* maintains an aggregate count of the number of events in an execution.

Rather than instrument at every point, these algorithms place code along select edges in the control-flow graph, by use of the spanning tree, with the guarantee that complete and accurate information can be ascertained for every point. This guarantee can be made because programs (usually) obey Kirchoff's flow law: the number of times execution enters a block is equal to the number of times execution leaves a block. We have shown that instrumenting control-flow edges rather than basic blocks is key to reducing the overhead introduced by instrumentation. Instrumenting edges rather than blocks gives more opportunity to place instrumentation code in areas of low execution frequency. Because placement of instrumentation code is driven by the spanning tree, selection of an appropriate spanning tree is also important in reducing overhead. We have given a simple heuristic for edge frequency that drives the spanning tree algorithms and is quite good at predicting areas of low execution frequency.

All the edge instrumentation problems can be characterized as cycle breaking problems, in which certain types of cycles in the control-flow graph must contain an edge with instrumentation code: all undirected cycles must be broken for edge profiling; all simple piped cycles must be broken for vertex profiling; all directed cycles and diamonds must be broken for tracing. We have shown how the cost of the optimal solutions to these various problems relate to one another (for a given control-flow graph and weighting) and presented a class of graphs for which optimal solutions to edge profiling are also optimal solutions to vertex profiling.

While the graph theoretic properties of these problems are important to understand, they must be tempered by experience in order to translate them into algorithms that can cope with programs found in practice. In practice, programs may not obey Kirchoff's flow law or obey standard calling conventions. We have shown how to profile programs that do not obey the flow law and have extended tracing to handle programs with multiple procedures.

Several open questions remain in the area of profiling. First, is there an efficient algorithm to optimally solve the vertex frequency problem with a set of edge counters or is the problem intractable? Second, are there better weighting schemes that can more accurately guide the placement of instrumentation code? Other interesting questions lie in the applications of the information produced by these tools. Once the profile or trace data has been collected, how does one present it in a way that helps the programmer understand how the program operates or where a performance bottleneck is?

Chapters 6 and 7 showed how the operations of slicing, differencing and integration can be extended to programs with complex control-flow. In particular, we have presented algorithms for slicing and computing the difference of programs with completely arbitrary control-flow and for integration of control-flow graphs with mostly reducible control-flow. The basis for the slicing and differencing algorithms lies in our result about flow/path-projections. We have shown that a control-flow graph H that is a flow/path-projection of a control-flow graph G has similar behavior to G: every vertex in H has equivalent behavior to its corresponding vertex in G. Backwards-closure along control and flow dependence edges in the program dependence graph is both necessary and sufficient for forming a flow/path-projection of a control-flow graph with respect to some vertex. We also defined the augmented control-flow graph in order to correctly determine when to include unconditional jumps in a program projection. It remains an open question whether there is an efficient algorithm for finding a program projection that is the minimal flow/path-projection of the original program under the standard control-flow translation.

Differencing uses the results about flow/path-projections to compare the behavior of vertices in different control-flow graphs. The differencing algorithm works for CFGs with completely arbitrary control-flow. Integration relies on the differencing operation in order to form a merged program dependence graph from the dependence graphs of the input programs. The main problem of integration is in reconstituting a control-flow graph from this merged graph. Reconstitution requires that certain ordering choices be made in order to produce a corresponding control-flow graph. We characterized how both control dependences and flow dependences force certain ordering choices and leave others open. Because the reconstitution algorithm is limited to producing a restricted set of CFGs, the integration algorithm is likewise limited and we cannot claim that it works for arbitrary control-flow graphs. Extending reconstitution so that it places no limitation on the structure of CFGs is highly desirable.

BIBLIOGRAPHY

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

2. B. Baker, "An algorithm for structuring flow graphs," *J. ACM* **24**(1) pp. 98-120 New York, NY, (January 1977).

3. T. Ball, S. Horwitz, and T. Reps, "Correctness of an algorithm for reconstituting a program from a dependence graph," Technical Report #947, Computer Sciences Department, University of Wisconsin – Madison, Madison, WI (July 1990).

4. T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages,* (Albuquerque, NM, January 19-22, 1992), pp. 59-70 ACM, (1992).

5. T. Ball and J. R. Larus, "Branch prediction for free," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (published in SIGPLAN Notices)* **28**(6) pp. 300-13 ACM, (June 1993).

6. T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *to appear in ACM Transactions on Programming Languages and Systems*, ACM, ().

7. U. Banerjee, *Dependence Analysis for Supercomputing,* Kluwer Academic Publishers, Boston, Massachusetts (1988).

8. D. Binkley, "Multi-procedure program integration," Ph.D. dissertation, Computer Sciences Department, University of Wisconsin, Madison, WI (1991).

9. D. Binkley, "Using semantic differencing to reduce the cost of regression testing," *Proceedings of the 1992 Conference on Software Maintenance*, pp. 41-50 (1992).

10. J. Cheng, "Slicing concurrent programs," *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging*, (May 1993).

11. J. D. Choi, B. P. Miller, and R. H. B. Netzer, "Techniques for debugging parallel programs with flowback analysis," *ACM Transactions on Programming Languages and Systems* **13**(4) pp. 491-530 (October 1991).

12. J. D. Choi and J. Ferrante, "What is in a slice," Unpublished draft, IBM T.J. Watson Research Center (December 1992).

13. R. F. Cmelik, S. I. Kong, D. R. Ditzel, and E. J. Kelly, "An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks," *ASPLOS-IV Proceedings (SIGARCH*

Computer Architecture News) **19**(2) pp. 290-302 (April 1991).

14. Systems Performance Evaluation Cooperative, *SPEC Newsletter (K. Mendoza, editor)* **1**(1)(1989).

15. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems* **13**(4) pp. 451-490 (October 1991 TOPLAS).

16. J. Ferrante and M. Mace, “On linearizing parallel code,” pp. 179-189 in *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages,* (New Orleans, LA, January 14-16, 1985), ACM, New York, NY (1985).

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

18. J. Ferrante, M. Mace, and B. Simons, “Generating sequential code from parallel code,” *Proceedings of the ACM 1988 International Conference on Supercomputing,* pp. 582-592 (July 1988).

19. J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, “Parallel processing: A smart compiler and a dumb machine,” *Proc. of the ACM SIGPLAN 1984 Symposium on Compiler Construction (SIGPLAN Notices)* **19**(6) pp. 37-47 (June 1984).

20. I. R. Forman, “On the time overhead of counters and traversal markers,” *Proceedings of the 5th International Conference on Software Engineering*, pp. 164-169 (March 1981).

21. K. B. Gallagher, “Using program slicing in software maintenance,” Ph. D. Thesis (Technical Report CS-90-05), University of Maryland, Baltimore County, Maryland (January 1990).

22. K. B. Gallagher, *private communication*. July 1993.

23. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Francisco (1979).

24. A. Goldberg, “Reducing overhead in counter-based execution profiling,” Technical Report CSL-TR-91-495, Stanford University, Stanford, CA (October, 1991).

25. A. J. Goldberg and J. L. Hennessy, “Mtool: An integrated system for performance debugging shared memory multiprocessor applications,” *IEEE Transactions on Parallel and Distributed Systems* **4**(1) pp. 28-40 (January 1993).

26. S. L. Graham, P. B. Kessler, and M. K. McKusick, “An execution profiler for modular programs,” *Software—Practice and Experience* **13** pp. 671-685 (1983).

27. J. K. Hollingsworth and B. P. Miller, "Dynamic control of performance monitoring on large scale parallel systems," Technical Report #1133, University of Wisconsin—Madison (January 1993).

28. S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," Report 690, Department of Computer Sciences, University of Wisconsin—Madison (March, 1987).

29. S. Horwitz, J. Prins, and T. Reps, "On the adequacy of program dependence graphs for representing programs," pp. 146-157 in *Conference Record of the 15th ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York (1988).

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

31. S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (published as SIGPLAN Notices)* **25**(6) pp. 234-245 ACM, (June 20-22, 1990).

32. S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 26-60 (January 1990).

33. S. Horwitz and T. Reps, "Efficient comparison of program slices," *Acta Informatica* **28** pp. 713-732 (1991).

34. S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," pp. 392-411 in *Proceedings of the Fourteenth International Conference on Software Engineering,* (May 11-15, 1992, Melbourne, Australia), ACM, New York, NY (1992).

35. J.W. Hunt and M.D. McIlroy, "An algorithm for differential file comparison," Report 41, Bell Laboratories, Murray Hill, N.J. ().

36. J. C. Hwang, M. W. Du, and C. R. Chou, "Finding program slices for recursive procedures," *Proceedings of IEEE COMPSAC 88,* (Chicago, IL, Oct. 3-7, 1988), pp. 220-227 IEEE Computer Society, (1988).

37. J. L. Kennington and R. V. Helgason, *Algorithms for Network Programming,* Wiley-Interscience, John Wiley and Sons, New York (1980).

38. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall Software Series, Englewood Cliff, NJ (1978. Second edition, 1988).

39. P. B. Kessler, “Fast breakpoints: Design and implementation,” *Proceedings of the ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation (SIGPLAN Notices)* **25**(6) pp. 78-84 ACM, (June, 1990).

40. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* Addison-Wesley, Reading, MA (1968. Second Edition: 1973).

41. D. E. Knuth and F. R. Stevenson, “Optimal measurement points for program frequency counts,” *BIT* **13** pp. 313-322 (1973).

42. B. Korel, “PELAS—Program Error-Locating Assistant System,” *IEEE Transactions on Software Engineering* **SE-14**(9) pp. 1253-1260 (September 1988).

43. W. Landi, B. G. Ryder, and S. Zhang, “Interprocedural modification side effect analysis with pointer aliasing,” *Proceedings of the ACM SIGPLAN ’93 Conference on Programming Language Design and Implementation (published in SIGPLAN Notices)* **28**(6) pp. 56-67 ACM, (June 1993).

44. J. R. Larus, “Abstract execution: A technique for efficiently tracing programs,” *Software—Practice and Experience* **20**(12) pp. 1241-1258 (December, 1990).

45. J. R. Larus, “Efficient program tracing,” *IEEE Computer* **26**(5) pp. 52-61 (May 1993).

46. J. R. Larus and T. Ball, “Rewriting executable files to measure program behavior,” *to appear in Software—Practice and Experience*, ().

47. S. Maheshwari, “Traversal marker placement problems are NP-complete,” Report No. CU-CS-092-76, Dept. of Computer Science, University of Colorado, Boulder, CO (1976).

48. S. McFarling, “Procedure merging with instruction caches,” *Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation,* (Toronto, June 26-28, 1991), *ACM SIGPLAN Notices* **26**(6) pp. 71-91 (June, 1991).

49. J. M. Mellor-Crummey and T. J. LeBlanc, “A software instruction counter,” *Third ASPLOS Proceedings (SIGARCH Computer Arch. News)* **17**(2) pp. 78-86 (April 3-6, 1989).

50. W. G. Morris, “CCG: A prototype coagulating code generator,” *Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation,* (Toronto, June 26-28, 1991), *ACM SIGPLAN Notices* **26**(6) pp. 45-58 (June, 1991).

51. D. Nachbar, “SPIFF - A program for making controlled approximate comparisons of files,” *Summer ’88 Usenix Conference*, ().

52. K.J. Ottenstein and L.M. Ottenstein, “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (Pittsburgh, PA, April 23-25,*

1984), ACM SIGPLAN Notices **19**(5) pp. 177-184 (May, 1984).

53. K. Pettis and R. C. Hanson, “Profile guided code positioning,” *Proceedings of the ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation (SIGPLAN Notices)* **25**(6) pp. 16-27 ACM, (June, 1990).

54. S. Pottle, *private communication*. October 1991.

55. R. L. Probert, “Optimal insertion of software probes in well-delimited programs,” *IEEE Transactions on Software Engineering* **SE-8**(1) pp. 34-42 (January, 1975).

56. W. Pugh, “A practical algorithm for exact array dependence analysis,” *Communications of the ACM* **35**(8) pp. 102-114 (August 1993).

57. C. V. Ramamoorthy, K. H. Kim, and W. T. Chen, “Optimal placement of software monitors aiding systematic testing,” *IEEE Transactions on Software Engineering* **SE-1**(4) pp. 403-410 (December, 1975).

58. M. V. S. Ramanath and M. Solomon, “Optimal Code for Control Structures,” pp. 82-94 in *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages,* (Albuquerque, NMP, ACM, New York (1982).

59. S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, “The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers,” *Proceedings of the 1993 ACM SIGMETRICS Conference*, (May 1993).

60. T. Reps and T. Teitelbaum, *The Synthesizer Generator: A system for constructing language-based editors,* Springer-Verlag, New York, NY (1988).

61. T. Reps and T. Bricker, “Illustrating interference in interfering versions of programs,” *Proceedings of the Second International Workshop on Software Configuration Management,* (Princeton, NJ, Oct. 24-27, 1989), *ACM SIGSOFT Engineering Notes* **17**(7) pp. 46-55 (November 1989).

62. T. Reps and W. Yang, “The semantics of program slicing and program integration,” in *Proceedings of the Colloquium on Current Issues in Programming Languages,* (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science*, Springer-Verlag, New York, NY (March 1989).

63. A. D. Samples, “Profile-driven compilation,” Ph. D. Thesis (Report No. UCB/CSD 91/627), University of California at Berkeley (April 1991).

64. V. Sarkar, “Determining average program execution times and their variance,” *Proceedings of the ACM SIGPLAN ’89 Conference on Programming Language Design and Implementation (SIGPLAN Notices)* **24**(7) pp. 298-312 ACM, (June 21-23, 1989).

65. B. Simons, B. Alpern, and J. Ferrante, "A foundation for sequentializing parallel code," *Proceedings of the Symposium on Parallel Algorithms and Architectures*, (July 1990).

66. A. J. Smith, "Cache memories," *ACM Computing Surveys* **14**(3) pp. 473-530 (1982).

67. MIPS Computer Systems, Inc., *UMIPS-V Reference Manual (pixie and pixstats),* MIPS Computer Systems, Sunnyvale, CA (1990).

68. R. E. Tarjan, *Data Structures and Network Algorithms,* Society for industrial and applied mathematics, Philadelphia, PA (1983).

69. D. W. Wall, "Predicting program behavior using real or estimated profiles," *Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation,* (Toronto, June 26-28, 1991), *ACM SIGPLAN Notices* **26**(6) pp. 59-70 (June, 1991).

70. M. Weiser, "Programmers use slices when debugging," *Communications of the ACM* **25**(7) pp. 446-452 (July 1982).

71. M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July, 1984).

72. W. Yang, "Identifying syntactic differences between two programs," *Software–Practice & Experience* **21**(7) pp. 739-755 (July 1991).

73. W. Yang, S. Horwitz, and T. Reps, "A program integration algorithm that accommodates semantics-preserving transformations," *Transactions on Software Engineering and Methodology* **1**(3)ACM, (July 1992).

TABLE OF CONTENTS