OPTIMALLY PROFILING AND TRACING PROGRAMS

by

Thomas Ball and James R. Larus

Computer Sciences Technical Report #1031

July 1991

Optimally Profiling and Tracing Programs

THOMAS BALL JAMES R. LARUS
tom@cs.wisc.edu larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin – Madison
1210 W. Dayton St.
Madison, WI 53706 USA

Technical Report #1031
Revision 1

September 6, 1991

An abridged version of this paper will appear in the 19th Symposium on Principles of Programming Languages (January 19-22, 1992).

ABSTRACT

This paper presents two algorithms for inserting monitoring code to profile and trace programs. These algorithms greatly reduce the cost of measuring programs. Profiling, which counts the number of times each basic block in a program executes, is widely used to measure instruction set utilization of computers, identify program bottlenecks, and estimate program execution times for code optimization. Instruction traces are the basis for trace-driven simulation and analysis and are used also in trace-driven debugging.

The profiling algorithm instruments a program for profiling by choosing a placement of counters that is optimized—and frequently optimal—with respect to the expected or measured execution frequency of each basic block and branch in the program. The tracing algorithm instruments a program to obtain a subsequence of the basic block trace—whose length is optimized with respect to the program's execution—from which the entire trace can be efficiently regenerated.

Both algorithms have been implemented and produce a substantial improvement over previous approaches, as the performance figures in this paper show. The profiling algorithm reduces the number of counters by a factor of two and the number of counter increments by a factor of three. The tracing algorithm reduces the file size and overhead of an already highly optimized tracing system by 20-40%.

CONTENTS

Optimally Profiling and Tracing Programs

THOMAS BALL JAMES R. LARUS
tom@cs.wisc.edu larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin - Madison
1210 W. Dayton St.
Madison, WI 53706 USA

1. INTRODUCTION

This paper presents two algorithms for inserting monitoring code to profile and trace programs. These algorithms greatly reduce the cost of measuring programs. Profiling, which counts the number of times each basic block in a program executes, is widely used to measure instruction set utilization of computers, identify program bottlenecks, and estimate program execution times for code optimization [2, 4, 6, 11, 13, 14, 17]. Instruction traces are the basis for trace-driven simulation and analysis and are used also in trace-driven debugging [9, 12, 18].

The product of this work is an *exact* basic block profile or trace—as opposed to the Unix *prof* command, which samples the program counter during program execution and does not produce exact measurements. This paper shows how to significantly reduce the cost of profiling and tracing with:

(1) an algorithm to instrument a program for profiling that chooses a placement of counters that is optimized—and frequently optimal—with respect to the expected or measured execution frequency of each basic block and branch in the program;

(2) an algorithm to instrument a program to obtain a subsequence of the basic block trace—whose length is optimized with respect to the program's execution—from which the entire trace can be efficiently regenerated.

Both algorithms have been implemented and substantially improve performance over previous approaches. The profiling algorithm reduces the number of counters by a factor of two and the number of counter increments by a factor of three. The tracing algorithm reduces the file size and overhead of an already highly optimized tracing system by 20-40%.

Each of these algorithms consists of two parts. The first chooses points in a program at which to insert profiling or tracing code. The second uses the results from the program's execution to derive a complete profile or trace. Surprisingly, the algorithms for instrumenting a program for profiling and tracing are identical and based on the well-known maximum spanning tree problem applied to the program's control-flow graph [20].

In the control-flow graph representation of a program, where a vertex represents a basic block of instructions and an edge represents passage of control from one block to another, instrumentation code can be

This work was supported in part by the National Science Foundation under grant CCR-8958530 and by the Wisconsin Alumni Research Foundation.

An abridged version of this paper will appear in the 19th Symposium on Principles of Programming Languages (January 19-22, 1992).

placed on vertices, edges, or some combination of the two. This work also shows that for both profiling and tracing, it is always better to place instrumentation code solely on edges.

The algorithms optimize placement of profiling and tracing code with respect to a *weighting* that assigns a nonnegative value to each edge in the control-flow graph. The cost of profiling or tracing a set of edges is proportional to the sum of the weights of the edges. The values in a weighting must satisfy Kirchoff's law of conservation of flow: the flow into a vertex is equal to the flow out of a vertex. A weighting captures the idea that every time control flows into a basic block, control also flows out of the block, and vice versa. Weightings can be obtained either by empirical measurement (*i.e.*, profiling) or a heuristic estimation. Our results show that a simple heuristic for estimating edge frequencies is accurate in predicting areas of low execution frequency at which to place instrumentation code. The placement algorithms are insensitive to the absolute values in the weighting. Instead, they rely on the order of edges induced by the weighting.

Our algorithms choose edges for instrumentation based on the control-flow of a program and a weighting. 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 (*i.e.*, via constant propagation). While there exist unstructured control-flow graphs for which the algorithms do not find an optimal placement, the algorithms optimize placements for a large class of well-structured control-flow graphs.

This paper has seven sections. The next section provides background material on control-flow graphs, weightings, and spanning trees. Section 3 shows how to efficiently profile programs and Section 4 describes how to efficiently trace programs. Section 5 presents results on the performance of the profiling and tracing algorithms. Section 6 reviews related work and Section 7 summarizes the paper and describes future work. The Appendix describes an algorithm for computing a weighting.

2. BACKGROUND

A control-flow graph (CFG) is a rooted directed graph $G = (V, E)$ with special vertex *EXIT* (distinct from the root vertex) that corresponds to a program in the following way: each vertex in V represents a basic block of instructions and each edge in E represents the transfer of control from one basic block to another. The root vertex represents the first basic block to execute and *EXIT* executes last. 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.

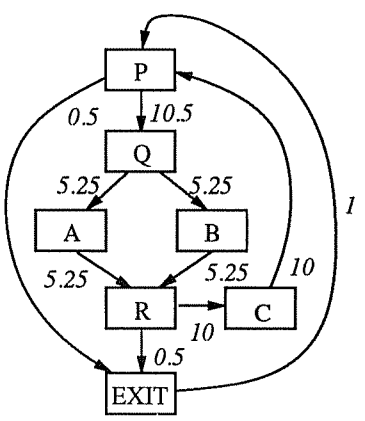
A vertex p is a *predicate* if there are distinct vertices a and b such that $p \rightarrow a$ and $p \rightarrow b$.

All *weightings* W of a CFG G assign a *nonnegative* value to every edge subject to the constraint that for each vertex v, the sum of the weights of edges with target v (the *incoming* edges of v) is equal to the sum of the weights of edges with source v (the *outgoing* edges of v). If vertex v has incoming edges with sources $y_1,...,y_m$ and outgoing edges with targets $z_1,...,z_n$, then v contributes the flow equation

$$\sum_{i=1}^{i=m} W(y_i \rightarrow v) = \sum_{j=1}^{j=n} W(v \rightarrow z_j)$$

where $W(x \rightarrow y)$ is the weight of edge $x \rightarrow y$. A weighting W assigns a value to every edge such that all flow equations ($|V|$ of them) are satisfied. The weight of a vertex is simply the sum of the weights of its

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



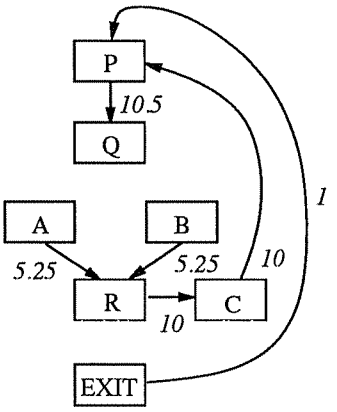


Figure 1. A program, its CFG with a weighting, and a maximum spanning tree. The edge *EXIT*→*P* is needed so that the flow equations for the root vertex (*P*) and *EXIT* are consistent. This edge does not correspond to an actual flow of control and is not instrumented.

incoming (or outgoing) edges. If W is a weighting of CFG G, then for a set of edges/vertices pl from CFG G, cost(G, pl, W) is the sum of the weights on the edges/vertices in pl.

A *spanning tree* of a directed graph G is a subgraph $G' = (V', E')$, where $V' = V$ and $E' \subseteq E$, such that for every pair of vertices (v,w) in G' there is a unique path (not necessarily directed) in G' that links v to w. A *maximum* spanning tree G' of graph G with weighting W is a spanning tree such that cost(G, E', W) is maximized. The maximum spanning tree for a graph can be computed efficiently by a variety of algorithms [20].

Figure 1 illustrates these definitions. The first graph is the CFG of the program shown—this graph has been given a weighting as shown by the values accompanying each edge. The second graph is a maximum spanning tree for the given weighting. Note that any vertex in the 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 by the path $C \rightarrow P \leftarrow EXIT$.

An *execution* EX of a CFG is a directed path that begins with the root vertex and ends with *EXIT* in which *EXIT* appears exactly once (we also refer to an execution as a sequence of vertices from such a directed path. If the CFG has multiple edges in the same direction between two vertices, collapse them into one edge and add their weights). 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. However, for any execution, the frequency of the edge *EXIT*→*root* is defined to be 1. The edge frequencies for any execution of a CFG constitute a weighting of the CFG.

3. 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 (vertex). The counter increments every time the block executes (pixie and other instrumentation tools use this

method [19]). There are two drawbacks to such an approach: too many counters are used and the total number of increments during an execution is larger than necessary.

The *vertex frequency* problem, denoted $VF(G, pl)$, is to determine a placement of counters pl in CFG G, where pl is a set of vertices and/or edges annotated with counters,[1] such that the frequency of each vertex in any execution of G can be deduced solely from the counters' values and the CFG G. Furthermore, to reduce the cost of profiling, these counters should be placed in areas of low execution frequency. That is, pl should solve $VF(G, pl)$ and minimize cost(G, pl, W) for a weighting W.[2] Such a pl is referred to as an optimal solution to $VF(G, pl)$ (with respect to weighting W).

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

To limit the number of permutations of these problems, pl is restricted to be a set of edges (*epl*) or a set of vertices (*vpl*). Section 3.2 shows that mixed placements (edges and vertices) are never better than pure edge solutions. We study the problems of optimally solving $VF(G, epl)$, $VF(G, vpl)$, and $EF(G, epl)$. Since there are CFGs for which there are no *vpl* solutions to $EF(G, vpl)$, it is not considered [15]. This section presents three results:

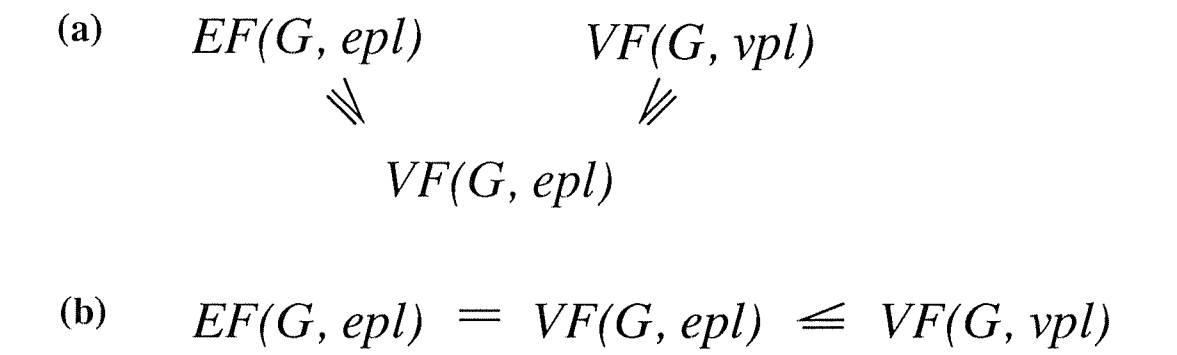


Figure 2. Case (a) shows the relationship between the costs of the optimal solutions of the three frequency problems for general CFGs. Case (b) shows the relationship when G is restricted to CFGs constructed from **while** loops, **if-then-else** conditionals, and **begin-end** blocks.

[1] A counter that appears on an edge $v \rightarrow w$ is incremented once every time control flows from vertex v to vertex w.

[2] We use a *weighting* as opposed to an arbitrary assignment of values because any execution of a CFG yields edge frequencies that form a weighting and because the sum of any two weightings is also a weighting. Thus, depending on how it is computed, a weighting can summarize many different executions of a given CFG. A weighting can be obtained by empirical measurement or a heuristic. The Appendix describes an algorithm for computing a good heuristic weighting. The absolute values in a weighting are not as important as the ordering of the vertices and edges induced by these values.

(1) an algorithm to optimally solve *EF* (*G*, *epl*) (Section 3.1);

(2) a comparison of the optimal solutions to *VF* (*G*, *epl*), *VF* (*G*, *vpl*), and *EF* (*G*, *epl*). Case (a) of Figure 2 summarizes the relationship between these three problems for general CFGs (Section 3.2);

(3) a proof that an optimal solution to *EF* (*G*, *epl*) is also an optimal solution to *VF* (*G*, *epl*) for a large class of structured CFGs (Section 3.3).

3.1. The Edge Frequency Problem

To solve *EF* (*G*, *epl*) by placing counters on edges, it is clearly sufficient to place a counter on the outgoing edges of each predicate vertex. However, this placement uses too many counters. From a well-known result in network programming, it follows that an edge-counter placement *epl* solves *EF* (*G*, *epl*) *iff* (*E*–*epl*) contains no cycle (possibly undirected) [7]. Since a spanning tree of a CFG represents a maximum size subset of edges without a cycle, it follows that *epl* is a minimum size solution to *EF* (*G*, *epl*) iff *E*–*epl* is a spanning tree of *G*. Thus, the minimum number of counters necessary to solve *EF* (*G*, *epl*) is $|E|-(|V|-1)$.

To see how such a placement solves the edge frequency problem, consider a CFG *G* and an *epl* such that *E*–*epl* is a spanning tree of *G*. Let each edge *e* in *epl* have an associated counter that is initially set to 0 and is incremented once each time *e* executes. If *v* is a leaf in the spanning tree (pick any vertex as the root), then all but one of the edges incident to *v* must be in *epl*. Since the edge frequencies for an execution are a weighting, 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*–*epl* are uniquely determined. If *E*–*epl* is not a spanning tree of *G* (*i.e.*, there is a cycle, possibly undirected, in

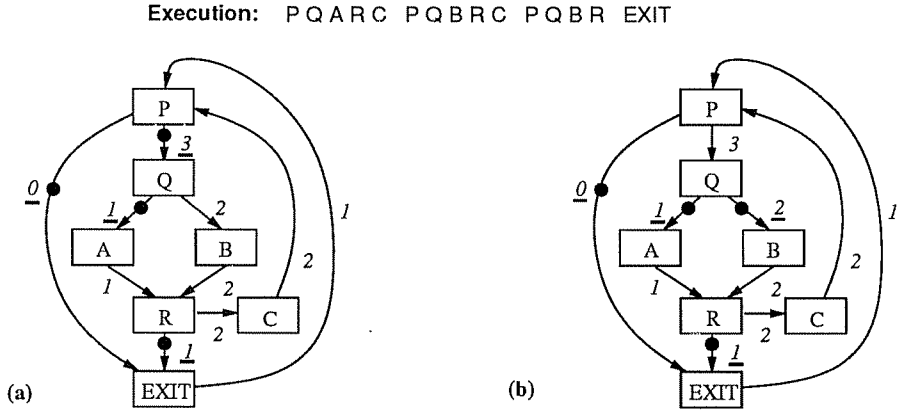


Figure 3. Solving *EF* (*G*, *epl*) using the spanning tree. The dotted edges are in *epl* and the remaining edges (*E*–*epl*) form a spanning tree of the CFG. The frequency of each edge in the execution is shown and the measured frequencies are underlined. For the weighting given in Figure 1, the *epl* in case (a) is not optimal (minimal) but the *epl* in case (b) is optimal.

E–epl), it can be shown that whenever the frequencies of edges in *epl* are fixed, there is more than one solution to the system of flow equations.

Case (a) of Figure 3 illustrates this process. The dotted edges in the CFG are the edges in *epl*. The other edges are in *E–epl* and form a spanning tree of the CFG. The edge frequencies are those for the execution shown. The *measured* frequencies are underlined. 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.

The propagation algorithm in Figure 4 does a post-order traversal of the spanning tree *E–epl* to propagate the frequencies of edges in *epl* to 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*, *epl*, *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. Therefore, the count for *e* is

```
global
    cnt: array[edge] of integer;          /* for each edge e in epl, cnt[e] = frequency of e in execution */

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

procedure DFS(G: CFG; epl: set of edges; v: vertex; e: edge)
let  IN(v) = { (w→v) | w→v ∈ E }  and  OUT(v) = { (v→w) | v→w ∈ E }  in
    in_sum := 0;
    for each e' ∈ IN(v) do
        if (e' ≠ e) and InSpanningTree(e') then DFS(G, epl, source(e'), e') fi
        in_sum := in_sum + cnt[e']
    od
    out_sum := 0;
    for each e' ∈ OUT(v) do
        if (e' ≠ e) and InSpanningTree(e') then DFS(G, epl, target(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 4. An algorithm to determine the frequencies of edges in the spanning tree *E–epl* given the frequencies of edges in *epl*. The algorithm uses a post-order traversal of the spanning tree. The predicate InSpanningTree(*e*) = ($e \in E–epl$).

found by subtracting the minimum of the two sums from the maximum.

For the weighting W given in Figure 1, the *epl* solution in case (a) of Figure 3 has cost(G, *epl*, W) = 16.75 and cost(G, $E-$*epl*, W) = 53.5–16.75 = 36.75. However, as case (b) of Figure 3 shows, there is an *epl* solution with cost(G, *epl*, W) = 11.5. This spanning tree has cost(G, $E-$*epl*, W) = 53.5–11.5 = 42. For this example, the *epl* placement of case (a) is suboptimal for any weighting. This is so because the cost of edge $P \rightarrow Q$ is equal to the cost of ($Q \rightarrow A + Q \rightarrow B$). Moving the counter from $P \rightarrow Q$ to $Q \rightarrow B$ incurs the cost ($Q \rightarrow A + Q \rightarrow B$) – ($P \rightarrow Q + Q \rightarrow A$), which can never be greater than zero.

It is clear that if *epl* solves *EF* (G, *epl*) and minimizes cost(G, *epl*, W), then $E-$*epl* is a maximum spanning tree of G. Any of the well-known maximum spanning tree algorithms described by Tarjan [20] will produce the maximum spanning tree of G with respect to weighting W. The edges that are not in the spanning tree (*epl*) solve *EF* (G, *epl*) and minimize cost(G, *epl*, W).

Although profiling has been described in terms of a single CFG, the algorithm scales up almost directly to programs with multiple procedures. The pre-execution spanning tree algorithm and post-execution propagation of edge frequencies are simply applied to each procedure separately. However, two problems can arise:

(1) If there is a CFG G with a directed path from *root* to *EXIT* that contains no edge in *epl* (which can occur only if *EXIT*→*root* is in *epl*), then there is a possible execution of G that increments no counter (since the edge *EXIT*→*root* is never traversed). Thus, it will be impossible to determine the exact count information for edges in G. To ensure that no such path arises, the maximum spanning tree algorithm can be seeded with the edge *EXIT*→*root*. In fact, for any CFG and weighting, there is always a maximum spanning tree that includes the edge *EXIT*→*root*. The derived count for the edge *EXIT*→*root* represents

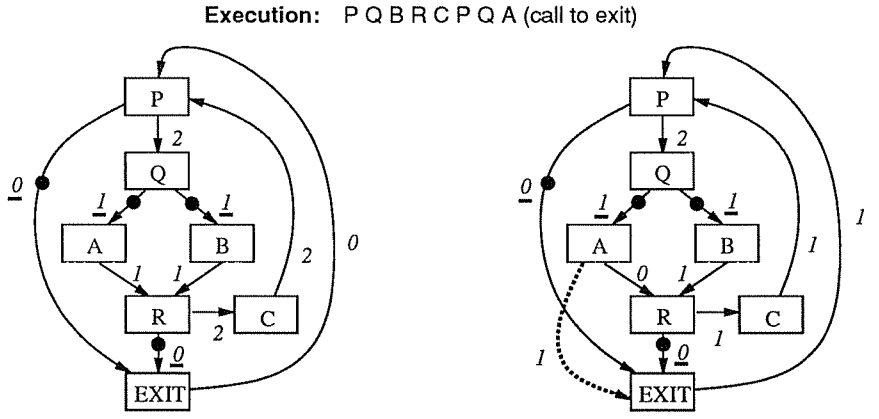


Figure 5. The first CFG executes the path shown and calls the exit routine at vertex A, which terminates the program. If the measured counts (underlined) are propagated to the spanning tree edges, incorrect values are computed. The second graph shows how this problem is solved. At program termination, the edge $A \rightarrow$*EXIT* is added and given count 1 to model the early termination of this procedure. After this edge has been added, the counts will be computed correctly.

the number of times the procedure G executed.

(2) The simple extension to multi-procedure profiling will determine the correct frequencies only if interprocedural control-flow occurs via procedure call and return and each call eventually has a corresponding return. Statically-determinable interprocedural jumps also fit in our framework. However, dynamically-computed interprocedural jumps (*e.g., setjmp/longjmp*) can cause problems. The common case of the call to the escape or exit procedure that terminates execution of the program illustrates this problem. In this case, the information on the activation stack at program termination is sufficient to correct the count error.

Suppose that the CFG in Figure 5 is called for the first time and executes the path shown. At vertex A, a call is made to the exit procedure. The measured counts are underlined. Propagating the measured counts to the edges in the spanning tree yields frequencies that are clearly incorrect for some edges. Edge $R \rightarrow C$ has a count of 2 yet was traversed only once.

Eliminating this error requires an examination of the activation stack at program termination. For each procedure X on the activation stack, add an edge $P_X \rightarrow EXIT_X$ to the CFG for procedure X, where P_X is the vertex from which procedure X calls the next procedure, and give it a count of 1. This edge models the termination of each active procedure. The propagation algorithm then can be applied to yield the correct results for each CFG. The second CFG in Figure 5 shows the added edge from $A \rightarrow EXIT$ and the derived counts.

3.2. Comparing the Three Frequency Problems

This section examines the relationships between the optimal solutions to $VF(G, epl)$, $VF(G, vpl)$, and $EF(G, epl)$ for general CFGs, as summarized in case (a) of Figure 2.

We first consider why $VF(G, epl)$, $VF(G, vpl)$, and $EF(G, epl)$ are the most interesting problems to study. Suppose that pl contains a mix of vertices and edges and optimally solves $VF(G, pl)$ or $EF(G, pl)$ for CFG G with weighting W. For any vertex v in pl, v's counter can be "pushed" off of v onto each outgoing edge of v, resulting in placement pl'. Since the cost of a vertex is equal to the sum of the costs of its outgoing edges, and some of v's outgoing edges may be in pl, $\text{cost}(G, pl', W) \leq \text{cost}(G, pl, W)$. Furthermore, pl' clearly solves the same problem as pl since no vertex or edge frequency information is lost in going from pl to pl'. Thus, for any CFG G and weighting W, a "mixed" solution to one of the problems can never be better than an optimal *epl* solution to the same problem.

It follows directly from this argument that for any CFG G and weighting W, the optimal solution to $VF(G, vpl)$ can never be better than the optimal solution to $VF(G, epl)$. It is easy to find examples where the optimal solution to $VF(G, epl)$ is better than the optimal solution to $VF(G, vpl)$.

Since any *epl* solution to $EF(G, epl)$ must also solve $VF(G, epl)$, it is clear that the optimal solution to $EF(G,epl)$ can never be better than the optimal solution to $VF(G, epl)$ for any CFG G and weighting W. As Figure 6 illustrates, there are cases where the optimal solution to $VF(G, epl)$ is better than the optimal solution to $EF(G, epl)$. In case (a), $E-epl$ is a maximum spanning tree of G and *epl* has cost 22. In case (b), *epl* has cost 20 and $E-epl$ is not a spanning tree because there is a cycle in $E-epl$ containing the inner four vertices. To see that this *epl* solves $VF(G, epl)$, note that the counts of the checked edges are uniquely determined by the counts of edges in *epl*. The count of each vertex's incoming or outgoing edges can be determined, which is sufficient to derive each vertex's count. The counts for the four edges in the inner cycle are not uniquely determined. For example, in order from left to right, one could assign the

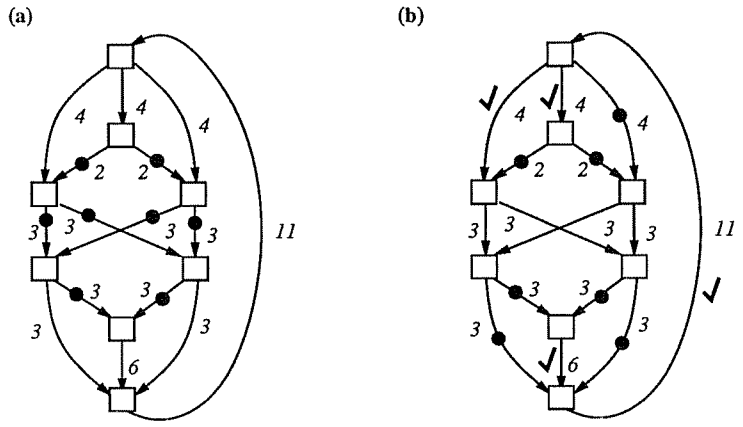


Figure 6. An example of a CFG and a weighting for which the optimal solution to *VF* (*G*, *epl*) is better than the optimal solution to *EF* (*G*, *epl*). The dotted edges are in *epl*. The first graph shows the optimal solution to *EF* (*G*, *epl*). The edges in *E*–*epl* form a maximum spanning tree of the graph. The lower cost *epl* placement in the second graph does not solve *EF* (*G*, *epl*) (as there is a cycle in *E*–*epl*) but does solve *VF* (*G*, *epl*). To see this, note that the count for each checked edge can be derived from the dotted edges and that this yields enough edge counts to determine the count for every vertex.

counts (4,2,2,4), (1,5,5,1), or (3,3,3,3) to these edges to satisfy the vertex flow equations.

The only examples that we have encountered in which *VF* (*G*, *epl*) betters *EF* (*G*, *epl*) exhibit unstructured control-flow such as found in Figure 6. For the CFG in Figure 1, the optimal solution to *EF* (*G*, *epl*) is also an optimal solution to *VF* (*G*, *epl*). Section 3.3 describes a class of graphs for which an optimal solution to *EF* (*G*, *epl*) is an optimal solution to *VF* (*G*, *epl*).

Finally, in comparing *EF* (*G*, *epl*) and *VF* (*G*, *vpl*) (for general CFGs), there are examples in which one is better than the other and vice versa. Case(b) of Figure 6 can be easily modified to show an example where *VF* (*G*, *vpl*) is better than *EF* (*G*, *epl*): simply consider each black dot as a vertex in its own right and split the dotted edge into two edges. The dots constitute the set *vpl* and solve *VF* (*G*, *vpl*) with cost 20. The optimal *epl* solution to *EF* (*G*, *epl*) for this graph still has cost 22.

There are many examples of structured CFGs where *EF* (*G*, *epl*) is preferable to *VF* (*G*, *vpl*). Consider the CFG in Figure 1 again. The vertex frequencies in this graph are related by the equations $Q = R = A + B$ and $EXIT = (P+R) - (C+Q)$. From these equations and the weighting in Figure 1, it turns out that the optimal solution to *VF* (*G*, *vpl*) is { *A*, *B*, *C*, *EXIT* }, with a cost of 21.5. The optimal solution to *EF* (*G*, *epl*) has cost 11.5. By instrumenting edges instead of vertices, there is greater freedom to pick and choose lower cost points (|E| as opposed to |V|).

3.3. Optimality Revisited

The previous section points out that the optimal way to solve the *VF* (*G*, *pl*) is to optimally solve *VF* (*G*, *epl*). Unfortunately, *VF* (*G*, *epl*) is a hard problem to solve optimally! We have made some

progress towards understanding this problem but have no efficient algorithm or proof of intractability for it yet. However, we believe that for most CFGs encountered in practice, an optimal solution to *EF* (*G*, *epl*) will provide an optimal (or near-optimal) solution to *VF* (*G*, *epl*). This section describes a class of CFGs for which an optimal solution to *EF* (*G*, *epl*) is also an optimal solution to *VF* (*G*, *epl*). The class of CFGs generated by **while** loops, **if-then-else** conditionals, and **begin-end** blocks is properly contained in this class.

Definition. A *diamond* consists of two simple directed paths (a path is simple if no vertex appears in it more than once) $PTH_a = p \rightarrow a \rightarrow \cdots \rightarrow z$ and $PTH_b = p \rightarrow b \rightarrow \cdots \rightarrow z$ such that $a \neq b$, and p and z are the only vertices common to both PTH_a and PTH_b.

THEOREM 3.1. If *epl* solves *VF* (*G*, *epl*), then *E*−*epl* contains no diamond or directed cycle.

PROOF. By contradiction. Suppose the antecedent and that (*E*−*epl*) contains a directed cycle *C*. Let execution EX_N consist of the following paths in order: a path PTH_1 from *root* to *v* such that *v* is in the cycle *C*; *N* traversals of cycle *C* from *v* to *v*; a path PTH_3 from *v* to *EXIT*. Let EX_{N+1} be the execution: PTH_1; *N*+1 traversals of cycle *C* from *v* to *v*; PTH_3. Since the cycle *C* contains no counters, the counts measured by *epl* do not differentiate these two executions, which have different frequencies for vertex *v*. Therefore, *epl* cannot solve *VF* (*G*, *epl*).

Now, suppose the antecedent and that (*E*−*epl*) contains a diamond *D* comprised of the paths $PTH_a = p \rightarrow a \rightarrow \cdots \rightarrow z$ and $PTH_b = p \rightarrow b \rightarrow \cdots \rightarrow z$. Let execution EX_a consist of the following paths: a path PTH_1 from the *root* vertex to *p*; PTH_a; a path PTH_3 from *z* to the *EXIT* vertex. Let EX_b be the execution: PTH_1; PTH_b; PTH_3. Since neither PTH_a nor PTH_b contains any counters, the counts measured by *epl* do not differentiate these two executions, which have different frequencies for vertices *a* and *b*. Therefore, *epl* cannot solve *VF* (*G*, *epl*). □

COROLLARY 3.2. For any CFG *G* with weighting *W*, an optimal *epl* solution to *VF* (*G*, *epl*) can never cost less than a minimal cost *epl* such that *E*−*epl* contains no directed cycle or diamond.

Consider the CFG in Figure 1 and any simple cycle (a cycle with *N* vertices is simple if *N*−1 of the vertices in the path representing the cycle are unique) in the graph. The cycle need not be directed. Each such cycle is either a directed cycle or a diamond. Let G^* represent all CFGs in which the only simple cycles are directed cycles or diamonds. For any CFG *G* in G^* with weighting *W*, the following two statements are equivalent:

(1) *epl* is a minimal cost set of edges such that *E*−*epl* contains no directed cycles or diamonds.

(2) *E*−*epl* is a maximum spanning tree.

Corollary 3.2, together with this result, implies that for any CFG *G* in G^* with weighting *W*, an optimal solution to *VF* (*G*, *epl*) can never be better than an optimal solution to *EF* (*G*, *epl*). Therefore, for this class of CFGs, an optimal solution to *EF* (*G*, *epl*) is an optimal solution to *VF* (*G*, *epl*).

The class of graphs G^* contains many examples of CFGs with multiple exit loops (such as in Figure 1), CFGs that require **gotos**, and even some irreducible graphs. However, in general, CFGs generated by programs with **repeat** loops or breaks are not always members of G^*. Take, for example, the program fragment in case (a) of Figure 7. The hashed edges form a cycle that is neither a directed cycle nor a diamond. Any CFG generated solely by **begin-end** blocks and **if-then-else** conditionals clearly is a member of G^*.

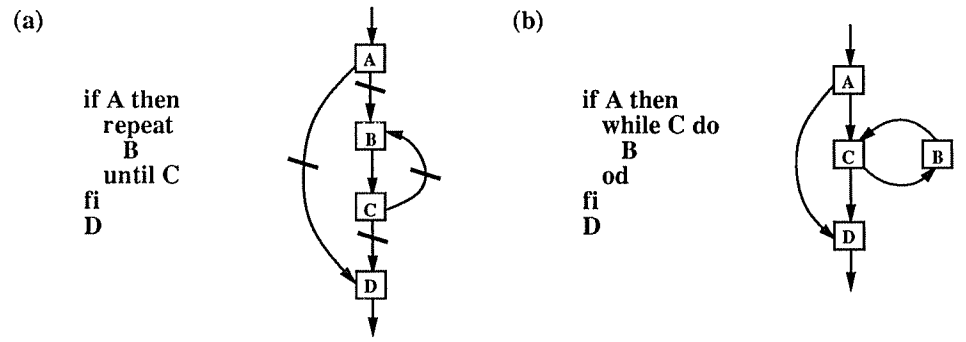


Figure 7.

The reason a repeat loop causes trouble is that the targets of the repeat predicate are both directly accessible from outside the loop. However, in the case of a while loop, only the exit edge of the while predicate is directly accessible from outside the loop (see case(b) of Figure 7). This is why the set of CFGs generated by **while** loops, **if-then-else** conditionals, and **begin-end** blocks is a subset of G^*.

To date, we have not found any examples of CFGs generated by structured programs with multi-exit loops for which there exists a solution to $VF(G, epl)$ with lower cost than an optimal solution to $EF(G, epl)$. Further work is required to find other classes of CFGs for which the optimal solutions to these problems are the same.

4. 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 basic blocks executed by the program. 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 that it writes a unique mark (witness) to a trace file whenever it executes. 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 [9]. 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 that there is a standard representation for witnesses (*i.e.*, a byte, half-word, or word per witness), then the tracing problem can be solved with significantly less time and storage overhead than either solution by writing witnesses when edges are traversed (not when vertices are executed) and carefully choosing the edges that write witnesses. Section 4.1 formalizes the trace problem for single-procedure

programs and shows that any *epl* solution to *EF* (*G*, *epl*) or *VF* (*G*, *epl*) also solves the tracing problem. Section 4.2 considers the complications introduced by multi-procedure programs. Section 4.3 illustrates how the tracing problem changes when the representation for witnesses depends on the number of witnesses used.

4.1. Single-Procedure Tracing

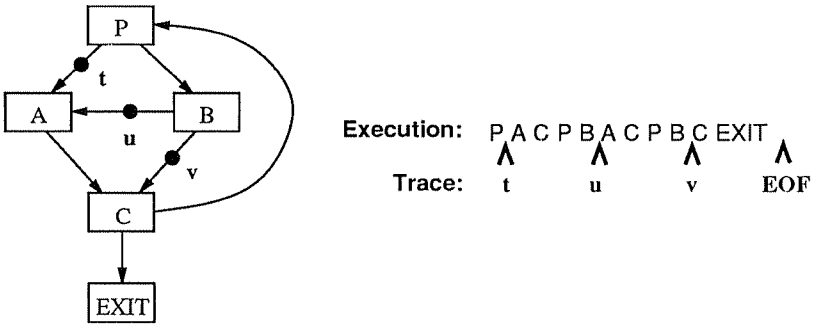
In this section, assume that basic blocks do not contain any calls and that the extra edge *EXIT*→*root* is not included in the CFG. The set of instrumented edges in the CFG is denoted by *epl*. In this application, whenever an edge in *epl* is traversed, a "witness" to that edge's execution is written to a trace file. No two edges in *epl* generate the same witness. 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 *epl* iff no edge in the path is in *epl*.

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

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

Figure 8 illustrates the above definitions. The edges marked by dots are in *epl* and the witnesses are shown next to them. Vertex *A* does not have a witness set because it is not a predicate. For the execution



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

Figure 8. Example of a traced function. Vertices *P*, *B*, and *C* are predicates and *EXIT* is the function return. The witnesses are shown by dots along edges. For the execution shown, the traced generated is **(t, u, v, EOF)**. The execution can be reconstructed from the trace using the witness sets to guide which branches to take.

shown, the trace generated is (**t, u, v, EOF**). The witness *EOF* is always the last witness in the trace.

Let us examine how the execution in Figure 8 can be regenerated from this trace. Re-execution starts at predicate P, the root vertex. To determine the successor of P, we read witness **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 since 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 from the trace. As witness **u** is a member of *witness*(C,P) but not *witness*$(C,EXIT)$, 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, epl, p, a) and *witness*(G, epl, p, b), where $a \neq b$, then two *different* executions of G generate the same trace file, which makes regeneration based solely on the control-flow and trace information impossible. For example, in Figure 8, if the edge $P \rightarrow A$ does 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 that write witnesses, *epl*, solves the *tracing problem* for CFG G, denoted *TP*(G, epl), iff for each predicate p in G with successors $q_1, ..., q_m$, for all pairs (q_i, q_j) such that $i \neq j$,

$$witness(G, epl, p, q_i) \cap witness(G, epl, p, q_j) = \varnothing$$

A witness placement *epl* and an execution *EX* of CFG G determine a trace as follows: let trace_record$(EX, epl) = (w_1, ..., w_k) \parallel EOF$, where w_i is the witness generated by the i^{th} edge in *EX* that is a member of *epl*. Given the CFG G, a set of edges *epl* that solves *TP*(G, epl), and trace_record(EX, epl), the algorithm in Figure 9 regenerates the execution *EX*. The following theorem captures the correctness of this algorithm:

THEOREM 4.1. If *epl* solves *TP*(G, epl) then for any execution *EX* of G, the call regenerate(G, *epl*, trace_record(EX, *epl*)) outputs the execution *EX*.

PROOF. We prove by induction that, at the beginning of the i^{th} loop of algorithm, the following invariant (*INV*) holds: the prefix *EX*$(1..i)$ of *EX* has been generated, *pc* is equal to *EX*(i), and one of the following is true: (*WIT* 1) the prefix trace_record$(EX(1..i), epl)$ of *trace* has been consumed and $wit = NULL$, or (*WIT* 2) the prefix (trace_record$(EX(1..i), epl) \parallel w$) of *trace* has been consumed and $wit = w$.

The fact that the algorithm terminates follows directly from the above invariant. Since the loop only terminates when $pc = EXIT$, the *EXIT* vertex occurs last in *EX* (and in no other place), and the invariant guarantees that at the i^{th} iteration of the loop the variable *pc* equals the i^{th} vertex of *EX*, the algorithm is guaranteed to loop once more when $i < |EX|$ and to terminate when $i = |EX|$.

Base Case: $i = 1$. The invariant *INV* is satisfied as $pc = root$, the *root* vertex has been output, $wit = NULL$, and none of the trace has been consumed.

Induction Hypothesis: at the beginning of the N^{th} iteration ($N > 1$) the invariant *INV* holds.

Induction Step: consider the beginning of the N^{th} iteration of the algorithm. By the Induction Hypothesis, the algorithm has generated the prefix *EX*$(1..N)$ of *EX* and $pc = EX(N)$. If *pc* is the *EXIT* vertex then the

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

Figure 9. Algorithm for regenerating an execution from a trace.

algorithm terminates, having generated the execution *EX*. If *pc* is not the *EXIT* vertex, then the loop iterates once more. The invariant holds at the beginning of the $N+1^{th}$ iteration by case analysis on *pc* at the beginning of the N^{th} iteration:

(1) Suppose *pc* is not a predicate vertex. The algorithm sets *newpc* to the control-flow successor of *pc*. Therefore, at the end of the loop, the prefix $EX(1..N+1)$ of *EX* has been generated and $pc = EX(N+1)$. By the Induction Hypothesis, at the beginning of the N^{th} iteration, either *WIT* 1 or *WIT* 2 holds. Furthermore, either $pc \rightarrow newpc$ is a member of *epl* or not. Thus, there are four subcases to consider:

(A) Suppose *WIT* 1 holds at *N* and $pc \rightarrow newpc \in epl$. Since trace_record($EX(1..N)$, *epl*) of *trace* has been consumed and trace_record($EX(1..N+1)$, *epl*) contains one more witness, the next witness in the trace file is the witness generated by $pc \rightarrow newpc$. Since $wit = NULL$, this witness is read and *WIT* 1 holds at $N+1$.

(B) Suppose *WIT* 1 holds at *N* and $pc \rightarrow newpc \notin epl$. It is clear that *WIT* 1 holds at $N+1$.

(C) Suppose *WIT* 2 holds at *N* and $pc \rightarrow newpc \in epl$. Since (trace_record($EX(1..N)$, *epl*) || *w*) = trace_record($EX(1..N+1)$, *epl*), *WIT* 1 holds at $N+1$.

(D) Suppose *WIT* 2 holds at *N* and $pc \rightarrow newpc \notin epl$. It is clear that *WIT* 2 holds at $N+1$.

(2) Suppose pc is a predicate vertex. By the Induction Hypothesis, at the beginning of the N^{th} iteration, either $WIT1$ or $WIT2$ holds.

(A) Suppose that $WIT1$ holds at N. Consider the suffix $EX(N..|EX|)$ of execution EX and the first witness w generated along this path. Note that $pc = EX(N)$. Let $q = EX(N+1)$. Since $wit = NULL$ and the prefix trace_record($EX(1..N)$, epl) of $trace$ has been read, w is read and assigned to variable wit. Witness w is a member of $witness(G, epl, pc, q)$ since w is the first witness generated on the path $EX(N..|EX|)$. Since epl solves $TP(G, epl)$, there can be no r ($r \neq q$) such that w is also a member of $witness(G, epl, pc, r)$. Therefore, at the end of the N^{th} iteration, the prefix $EX(1..N+1)$ has been generated and $pc = EX(N+1)$. If $pc \rightarrow newpc \in epl$ then $WIT1$ holds at $N+1$. On the other hand, if $pc \rightarrow newpc \notin epl$ then $WIT2$ holds at $N+1$.

(B) Suppose that $WIT2$ holds at N. Since the prefix (trace_record($EX(1..N)$, epl) $\|$ w) of $trace$ has been consumed, w is the first witness generated by the suffix $EX(N..|EX|)$ of execution EX. Again, let $q = EX(N+1)$. Witness w is a member of $witness(G, epl, pc, q)$. Since epl solves $TP(G, epl)$, at the end of the N^{th} iteration the prefix $EX(1..N+1)$ of EX has been generated and $pc = EX(N+1)$. If $pc \rightarrow newpc \in epl$, then $WIT1$ holds at $N+1$. If $pc \rightarrow newpc \notin epl$ instead, then $WIT2$ holds at $N+1$. □

The following theorem shows that epl solves $TP(G, epl)$ exactly when the set of edges $E-epl$ contains no diamond or directed cycle. This result implies that any epl that solves $EF(G, epl)$ also solves $TP(G, epl)$. Therefore, if the set of edges T is a maximum spanning tree of G, $epl = E-T$ solves $TP(G, epl)$. Also, Theorem 3.1 implies that any epl that solves $VF(G, epl)$ solves $TP(G, epl)$.

THEOREM 4.2. The set $E-epl$, where E represents the edges of CFG G and $epl \subseteq E$, contains no directed cycles or diamonds iff epl solves $TP(G, epl)$.

PROOF. By contradiction.

$\rightarrow$ Suppose that there exists a predicate p with distinct successors a and b such that $\{w\} \subseteq witness(G, epl, p, a) \cap witness(G, epl, p, b)$. Let $x \rightarrow y$ be the edge in epl that generates witness w. There is a witness-free path $PTH_a = p \rightarrow a \rightarrow \cdots \rightarrow x$ and a witness-free path $PTH_b = p \rightarrow b \rightarrow \cdots \rightarrow x$. If any vertex in PTH_a (PTH_b) occurs more than once in PTH_a (PTH_b), then there is a directed cycle in $E-epl$. Suppose that PTH_a and PTH_b are both simple. Let z be the first vertex in PTH_a besides p that is shared by PTH_b. There is a diamond in $E-epl$ consisting of the prefix of PTH_a from p to z and the prefix of PTH_b from p to z.

$\leftarrow$ Suppose that there is a diamond or a directed cycle in $E-epl$. First, consider the case of a diamond consisting of the paths $PTH_a = p \rightarrow a \rightarrow \cdots \rightarrow x$ and $PTH_b = p \rightarrow b \rightarrow \cdots \rightarrow x$. By the definition of a diamond, both paths are simple and share only the vertices p and x. Consider any path from x to the $EXIT$ vertex. If the path contains no edges in epl then EOF is a member of both $witness(G, epl, p, a)$ and $witness(G, epl, p, b)$. Instead, suppose w is the witness generated by the first edge in this path that is in epl. Witness w is a member of both $witness(G, epl, p, a)$ and $witness(G, epl, p, b)$.

Suppose that there is a directed cycle in $E-epl$. At least one of the vertices in this cycle is a predicate since there is a path from every vertex in G to the $EXIT$ vertex and there can be no directed

cycle that contains the *EXIT* vertex (as there are no outgoing edges from *EXIT*). Let $p \rightarrow a$ be an edge in the cycle where p is a predicate. Let $p \rightarrow b$ be another outgoing edge from p, where $a \neq b$. Since there is a witness-free path $p \rightarrow a \rightarrow \cdots \rightarrow p$, any witness that is in *witness*(G, epl, p, b) is also in *witness*(G, epl, p, a). □

The remaining open questions are whether an optimal solution to *VF*(G, epl) is an optimal solution to *TP*(G, epl), and whether optimally solving *TP*(G, epl) is an inherently intractable problem. Finding a minimum size set of edges *epl* such that $E - epl$ contains no directed cycles (Feedback Arc Set) or diamonds (Uniconnected Subgraph) are NP-complete problems that bear some similarities to *TP*(G, epl), but we have not succeeded in constructing a reduction [5, 10]. There are different constraints on *TP*(G, epl) that complicate matters. First, the problem is to find a minimum cost set of edges *epl* with respect to a weighting rather than one of minimum size. Second, the weighting is not an arbitrary assignment of values—all the values are nonnegative and satisfy the flow equations. Although optimally solving *TP*(G, epl) is probably an NP-complete problem, we should keep in mind that for any CFG G in G^* an optimal solution to *EF*(G, epl) is an optimal solution to *TP*(G, epl). We believe that an optimal solution to *EF*(G, epl) provides an optimal or near-optimal solution to *TP*(G, epl) for most CFGs found in practice.

4.2. Multi-Procedure Tracing

Unfortunately, tracing does not extend as easily to multiple procedures as does profiling. There are several complications that we illustrate with the CFG in Figure 8. Suppose that basic block B contains a call to procedure X and that execution starts at P and continues to B, where procedure X is called. After procedure X returns, suppose that C executes. This call creates problems for the regeneration process since the witnesses generated by procedure X, possibly an enormous number of them, precede the witness v in the trace file.

In order to determine which branch of predicate P to take, the witnesses generated by procedure X must be buffered or witness set information must be propagated interprocedurally. The first solution is impractical because there is no bound on the number of witnesses that may have to be buffered. The second solution eliminates the possibility of separate compilation (instrumentation) and is complicated by multiple calls to the same procedure from a procedure and by calls to unknown procedures. Furthermore, if witness numbers are reused in different procedures, which greatly reduces the amount of storage needed per witness, then the second approach becomes even more complicated.[3]

The following restrictions describe the tracing problem for programs with multiple procedures: (1) only one trace file is generated; (2) witnesses can be reused in different procedures but not within a procedure; (3) procedures and functions are separately instrumented (*i.e.*, no interprocedural analysis). The solution presented in this section places "blocking" witnesses that prevent all predicates in a CFG from "seeing" a basic block that contains a call site or from seeing the *EXIT* vertex in that CFG. This ensures that whenever the regenerator is in CFG G and reads a witness to determine which branch of a predicate to take, the

[3]If a separate trace file was 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.

witness is guaranteed to have been generated by an edge in G.[4]

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

Definition. The set $\{ epl_1, ..., epl_m \}$ solves the *tracing problem for a set of CFGs* $\{ G_1, ..., G_m \}$ iff, for all i, epl_i solves $TP(G_i, epl_i)$ and epl_i has the blocking property for G_i.

The regeneration algorithm in Figure 9 need only be modified to maintain a stack of currently active procedures: when the algorithm encounters a vertex with a call, 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 continues executing the caller from the point of the call. For multiple calls per vertex, the algorithm also has to keep track of the current instruction in the current basic block.

An easy way to ensure that *epl* has the blocking property is to include each incoming edge to a vertex that contains a call in *epl*, and to include each incoming edge to *EXIT* in *epl*. This approach is suboptimal for several reasons. Consider the first control-flow fragment in Figure 10 in which the dashed vertices contain calls and the black dots represent blocking witnesses. The witness on edge $H \rightarrow I$ is redundant since the

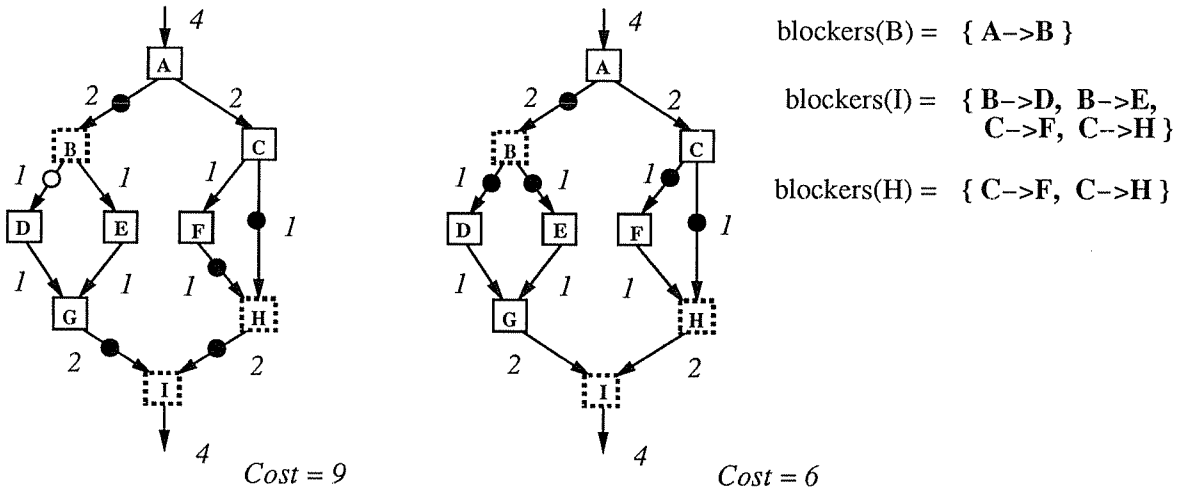


Figure 10. Two placements of blocking witnesses. The dashed vertices (*B*, *I*, and *H*) are call vertices. In the first subgraph, a blocking witness is placed on each incoming edge to a call vertex (black dots). This placement is suboptimal because the witness on edge $H \rightarrow I$ is not needed and because a witness must be added to edge $B \rightarrow D$ to solve the tracing problem (white dot). In the second subgraph, blocking witnesses are placed as far away from call vertices as possible, resulting in an optimal placement.

[4]In some tracing applications, data other than witnesses (such as addresses) are also written to the trace file. Vertices in the CFG that generate address information 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.

witnesses on edges $F \rightarrow H$ and $C \rightarrow H$ already block all paths from C to I. Also, the edge $B \rightarrow D$ requires a witness in order for *epl* to solve the tracing problem.

These problems can be solved by placing blocking witnesses as far away as possible from the vertices that they are meant to block. 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. One of the edges in this path must contain a witness to satisfy the blocking property. 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 *TP* (G, *epl*).

The second placement of blocking witnesses in Figure 10 uses the above approach. After blocking witnesses are added, no other witnesses need to be added to solve the tracing problem. However, in general, this is not always the case. Therefore, computing *epl* becomes a two step process: (1) place the blocking witness; (2) ensure that *TP* (G, *epl*) is solved by adding edges to *epl*. The details of the algorithm follow:

Definition. Let v be a vertex in CFG G. The *blocking edges* of v are defined as follows:

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

The following algorithm computes the optimal cost set of edges (for any weighting) that has the blocking property:

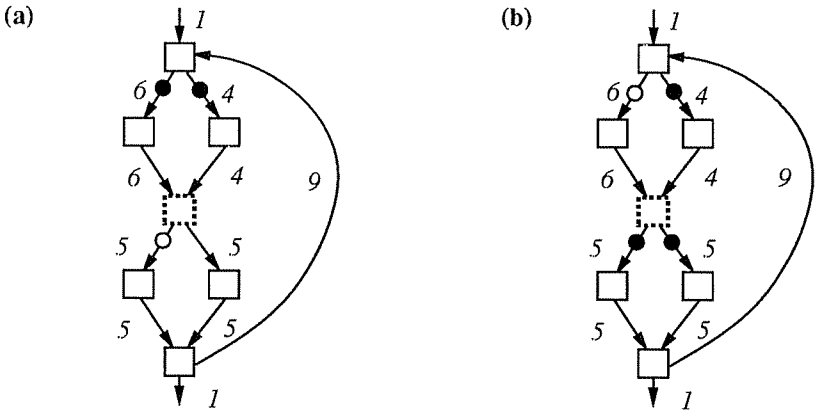


Figure 11. Example (a) illustrates the result from first finding blocking witnesses (black dots) and then applying the maximum spanning tree algorithm (white dot). Example (b) shows the suboptimal placement that results from applying these phases in reverse order. The spanning tree algorithm places the black dots. The white dot is added to block the topmost predicate from seeing the call vertex. This placement is suboptimal because one of the black dots on an edge with weight 5 could be removed.

```
epl := ∅;
for each vertex v ∈ { EXIT } ∪ { w | vertex w contains a call } do
    epl := epl ∪ blockers(G, v)
od
```

To ensure that *epl* also solves *TP* (*G*, *epl*), edges are added to *epl* so that *E*–*epl* contains no diamonds or directed cycles. The maximum spanning tree algorithm, modified so that no edge in *epl* is allowed in the spanning tree, is applied to *G*. If the edges that are not in the spanning tree are added to *epl*, then *epl* is guaranteed to have the blocking property *and* solve *TP* (*G*, *epl*).[5] Applying this algorithm to the control-flow fragment in case (a) of Figure 11, the blocking phase adds the black dot edges to *epl*. The spanning tree phase adds the white dot edge to *epl*.

One might question whether it is better to reverse the above process and first compute an *epl* that solves *TP* (*G*, *epl*) using the maximum spanning tree algorithm, adding blocking witnesses as needed afterwards. Case (b) of Figure 11 shows that this approach can yield undesirable results. The black dot edges are placed by the spanning tree phase and solve *TP* (*G*, *epl*) but do not satisfy the blocking property. The white dot edge must be added to satisfy the blocking property and creates a suboptimal *epl*.

4.3. Bit Twiddling

Suppose that a program contains predicates with only two successors (*i.e.*, no case statements are allowed) and that each outgoing edge of a predicate vertex generates a witness. While this approach records more witnesses than necessary, only one bit per witness is needed to distinguish the witness set at any predicate. If witnesses are placed on fewer edges (using the spanning tree approach), then some predicate branches will not generate witnesses. However, in general, more bits will be needed per witness in this case.

The CFGs in Figure 12 illustrate this tradeoff. 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.

The spanning tree approach has the advantage that it is sensitive to weightings. For example, if the edge with witness **c** had low probability of executing then the witness placement in case (a) would be superior to that in case (b).

On the other hand, it is easy to construct examples where the one-bit approach is superior. For instance, consider a loop similar to that in Figure 12 that contains a chain of nine diamonds. The spanning tree approach requires 11 distinct witnesses, resulting in 4 bits per witness. In the one-bit approach every iteration of the loop generates 10 bits of trace (one for each diamond in the loop plus one for the loop back-edge). In the spanning tree approach the number of bits per iteration ranges from 4 to 36. If every predicate has equal probability of taking either branch then, on average, each iteration generates 20 (4 + 32/2) bits of trace.

---

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

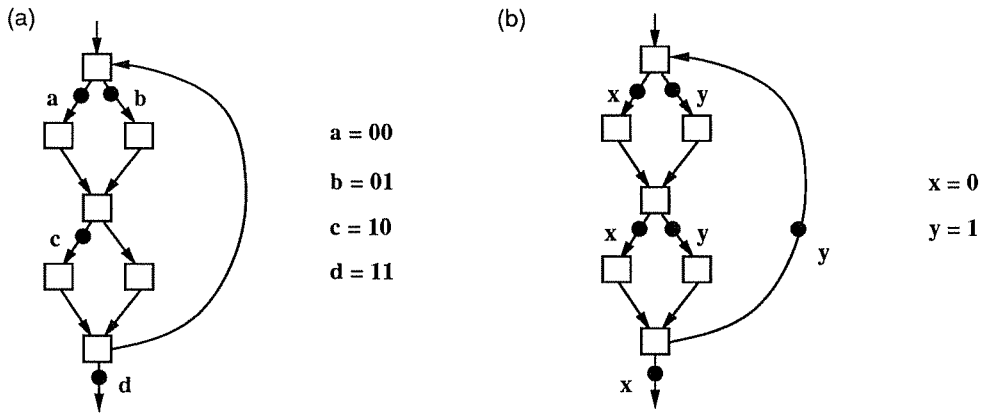


Figure 12. These CFGs illustrate the tradeoff between (a) placing witnesses according to the spanning tree approach and (b) placing witnesses on every outgoing edge of a predicate vertex. In general, the former approach requires more bits per witness and some edges do not generate witnesses. In the latter approach, every (predicate) edge generates a witness but only one bit per witness is needed.

Further study is needed to determine whether one of these approaches is better than the other in practice. In addition, the one-bit approach would have to be extended to handle predicates with more than two outgoing edges.

5. PERFORMANCE

This section describes several experiments that demonstrate that the algorithms presented above significantly reduce the cost of profiling and tracing real programs.

5.1. Profiling Performance

We implemented the counter placement algorithm for profiling in QP, which is a basic block profiler similar to MIPS's *pixie* [19]. QP can either insert counters in every basic block in a program (*slow* mode) or along the subset of edges identified by our algorithm (*quick* mode).

We used the SPEC benchmark suite to test QP [3]. 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. The timings were run on a DECstation 5000/200 with 96MB of main memory and local disks.

Table 1 shows the cost of running the benchmarks with profiling. The column labeled "Slow" contains the time for programs modified by QP to have a counter in each basic block. The column labeled "Quick" contains the times for programs with optimized edge profiling. The column labeled "Pixie" contains the times for programs profiled by *pixie*, which inserts a counter in each basic block. These times are less than the times required by slow QP because *pixie* rewrites the program to free 3 registers, which enables it to insert a code sequence that is about half the size of the one used by QP (6 instructions vs. 11 instructions). In fact, the *pixie* code sequence can be reduced to 5 instructions. The column labeled "Quick+" is the projected time for quick QP tracing using this 5 instruction code sequence. As can be seen, the placement algorithm reduces the overhead of profiling dramatically, from 11-424% to 9-105%. Fortunately, the greatest improvements occurred in programs in which the profiling overhead was largest, since these programs had more conditional branches and more opportunities for optimization.

SPEC	**Slow**		**Quick**		**Pixie**		**Quick+**	
Benchmark	(sec.)	%	(sec.)	%	(sec.)	%	(sec.)	%
gcc (C)	32.2	222.0	19.5	95.0	24.5	145.0	14.3	43.2
espresso (C)	71.5	177.1	45.6	76.7	52.6	103.9	34.8	34.9
spice	379.9	62.7	320.7	37.3	320.8	37.4	273.1	17.0
doduc	197.5	56.6	142.6	13.1	180.1	42.8	133.6	5.9
nasa7	1045.9	15.7	1025.9	13.5	992.4	9.8	959.3	6.1
li (C)	945.9	218.9	553.6	86.6	808.2	172.5	413.4	39.4
eqntott (C)	313.1	423.6	122.5	104.8	178.7	198.8	88.3	47.7
matrix300	311.5	13.6	308.8	12.6	292.4	6.6	290.0	5.7
fpppp	240.2	36.2	199.7	13.2	207.2	17.5	187.0	6.0
tomcatv	179.4	10.7	176.6	8.9	176.4	8.8	168.7	4.1

Table 1. Cost of profiling. For Slow profiling, QP inserts a counter in each basic block. For Quick profiling, QP inserts a counter along selected edges. Pixie is a MIPS utility that inserts a counter in each basic block. Quick+ is the time that Quick profiling would require if QP used the efficient *pixie* counter instruction sequence. The columns labeled % show the additional cost of profiling, with respect to the unprofiled program's execution time.

	Counter Increments					Dynamic
SPEC	**Slow**	**Quick**	**Slow/**	**Feedback**	**Slow/**	Block
Benchmark			**Quick**		**Feedback**	Size
gcc (C)	27149754	8458003	3.2	5324315	5.1	4.6
espresso (C)	91259523	33139589	2.8	27247737	3.3	5.0
spice	308194784	180543666	1.7	172595830	1.8	10.6
doduc	130897009	45651338	2.9	35920460	3.6	11.2
nasa7	298530617	254628038	1.2	251638412	1.2	30.2
li (C)	1208747235	413622801	2.9	289473770	4.2	4.1
eqntott (C)	465938460	114410157	4.1	112562938	4.1	2.3
matrix300	60035631	54951383	1.1	54947186	1.1	46.1
fpppp	25932871	6186762	4.2	4098093	6.3	100.8
tomcatv	35012274	27762776	1.3	21254823	1.6	56.3

Table 2. Reduction in counter increments due to optimized counter placement. The column labeled Slow is the number of increments in basic blocks. The column labeled Quick is the number of increments along edges chosen by the placement algorithm guided by the heuristic weighting described above. The column labeled Feedback records the number of increments along edges chosen by the placement algorithm using an exact weighting from a previous run. The last column is the average dynamic basic block size.

Table 2 shows this improvement in another way. It records the number of counter increments for both Slow and Quick profiling. 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 *fpppp* benchmark produced an interesting result. While it showed the greatest reduction in counter increments, the overhead for measuring every basic block was quite low at 36% and the average dynamic basic block size was 101. This implies that large basic blocks dominated the execution of *fpppp*. Thus, even though many basic blocks of smaller size executed (which yielded the reduction in counter increments), they contributed little to the running time of the program. The FORTRAN program *doduc*, while it has a dynamic block size of 10 instructions, has "an abundance of short branches" [3] that accounts for its reduction in counter increments. The decrease in run time overhead for *doduc* was substantial at 57%-13%.

For programs that frequently executed conditional branches, the improvements were large. For the 4 C programs (*gcc*, *espresso*, *li*, and *eqntott*), the placement algorithm reduced the number of increments by a factor of 3 and the overhead by a factor of 2-3.

Table 2 also demonstrates that the heuristic weighting algorithm described in the Appendix is good. The column labeled "Feedback" contains the number of counter increments when the placement algorithm was guided by an exact profile from a previous run of the same program with identical input. As can be seen, the difference in cost between the heuristic and exact weightings was usually small.

The cost of modifying a program to place counters along edges was higher than placing counters in each basic block, primarily because of the additional work required to compute a program's control-flow graph and to determine counter placement. However, the cost was quite reasonable (exact times omitted since QP has not yet been tuned).

5.2. Tracing Performance

The witness placement algorithm was also implemented in the AE program tracing system [9]. 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. The combined file requires the changes to the placement algorithm described in Section 4.2.

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. Improvement in the AE program tracing system from placing witnesses along edges. 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 witnesses. 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.

Table 3 shows the reduction in total file size ("File"), witness trace size ("Trace"), and execution time that result from switching from the original algorithm of recording each conditional ("Old") to a witness placement ("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 reduces the number of witnesses by a factor of 3 and 2.7 times.

6. RELATED WORK

This section describes related work on efficiently profiling and tracing programs.

6.1. The Knuth/Stevenson Algorithm

Knuth and Stevenson exactly characterize when a set of vertices *vpl* solves *VF* (*G*, *vpl*) and show how to compute the minimum size *vpl* that solves *VF* (*G*, *vpl*) [8]. They construct a graph *G′* from CFG *G* such that *vpl* solves *VF* (*G*, *vpl*) iff *epl′* solves *EF* (*G′*, *epl′*), where *vpl* can be derived from *epl′* by a one-to-one and onto map from edges in *G′* to vertices in *G* that falls out from the construction. The authors note that their algorithm can be modified very easily to compute a minimum cost *vpl* solution to *VF* (*G*, *vpl*) given a set of measured or guessed vertex frequencies.

As this paper shows, if counter placement is restricted to vertices, a minimum cost solution to the vertex frequency problem cannot always be found—lower cost solutions often can be obtained by placing counters on edges instead of vertices. Furthermore, since the optimal solution to *EF* (*G*, *epl*) is equal to the optimal solution to *VF* (*G*, *epl*) for a large class of CFGs and the optimal solution to *VF* (*G*, *vpl*) can never be better than *VF* (*G*, *epl*), we expect *EF* (*G*, *epl*) to perform better than *VF* (*G*, *vpl*) in practice.

6.2. The Insertion of Software Probes in Well-Delimited Programs

Probert discusses the problem of solving *EF* (*G*,*vpl*) [15], which is not always possible in general. Using graph grammars, he characterizes a set of "well-delimited" programs for which *EF* (*G*, *vpl*) can always be solved. This class of graphs arises by introducing "delimiter" vertices into well-structured programs. These extra vertices allow *EF* (*G*, *vpl*) to be solved for this class of graphs. Although Probert declares otherwise, delimiter vertices serve the same function as "artificial" vertices that are inserted on an edge: they allow edge frequencies to be determined from vertex frequencies.

Probert is also concerned with finding a minimal size set of measurement points as opposed to a minimal cost set of measurement points.

6.3. Profiling Using Control Dependence

Sarkar describes how to choose profiling points using control dependence and has implemented a profiling tool for the PTRAN system, which uses the profile information to guide the automatic parallelization of FORTRAN programs [17]. His algorithm finds a minimum size set of edges *epl* that solves *EF* (*G*, *epl*) based on a variety of rules relating control dependence and control-flow, as opposed to the spanning tree method given here. There are several other major differences between his work and the work reported here:

(1) The algorithm only works for a subclass of reducible CFGs. Reducible CFGs that do not fall into this subclass must be transformed for the algorithm to work properly. Our algorithm can be applied to any CFG.

(2) The algorithm does not use a weighting or other method to place counters at points of lower execution frequency. As a result, the algorithm may produce a suboptimal solution such as that in case (a) of Figure 3.

(3) When the bounds of a **DO** loop are constants, the algorithm will eliminate the loop iteration counter. Our analysis is based solely on the control-flow information from the program and does not make such optimizations.

6.4. Optimal Placement of Traversal Markers

Ramamoorthy, Kim, and Chen discuss the problem of instrumenting a single-procedure program with a minimal number of monitors so that the traversal of any path through the program may be ascertained after an execution [16]. 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 handle the problem of tracing multi-procedure programs.

The authors are interested in finding a minimal *size* solution to *TP*(*G*, *epl*), an NP-complete problem [10], and develop an efficient heuristic procedure for constructing a near minimal size solution. However, a minimum size solution does not necessarily yield a minimum cost solution; sometimes a lower cost solution can be obtained by instrumenting more lower cost points rather than fewer higher cost points. We expect the problem of finding a minimum cost solution to *TP*(*G*, *epl*) (where the edge values yield a weighting) to be NP-complete also.

7. SUMMARY AND FUTURE WORK

This paper introduced two algorithms for efficiently profiling and tracing programs. These algorithms optimize placement of instrumentation code with respect to a weighting of the control-flow graph. The placements for a large class of graphs are optimal, but there exist programs for which the algorithms produce suboptimal results.

Many interesting questions remain open. 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 other classes of graphs for which an optimal solution to the edge frequency problem is also an optimal solution to the vertex frequency problem? How are these problems related to the tracing problem and its optimal solution? Finally, can better weighting approximation algorithms be found?

As previously noted, the profiling algorithm has been implemented in a profiling tool called QP and the tracing algorithm is part of the AE tracing system [9]. Both tools run on several machines and are available from James Larus.

ACKNOWLEDGEMENTS

We would like to thank Susan Horwitz for her support of this work. Gary Schultz and Jonathan Yackel provided valuable advice on network programming. Eric Bach pointed out the NP-complete Feedback Arc problem. Samuel Bates, Paul Adams, and Phil Pfeiffer critiqued many descriptions of the work in progress. Thanks also to Guri Sohi and Tony Laundrie, who provided their code for a basic-block profiler that eventually became QP, and to Mark Hill, who provided the disk space and computing resources for the performance measurements. Chris Fraser's insightful comments on tracing led to the addition of Section 4.3.

APPENDIX - A WEIGHTING ALGORITHM

This section describes how to compute a weighting for reducible CFGs, those CFGs with single-entry loops. A CFG is reducible iff for each backedge e of G (as defined by a depth-first search from the root vertex), target(e) dominates source(e). Other equivalent characterizations of reducibility are given in [1]. The edge $EXIT \rightarrow root$ is not counted as a backedge even though it is identified as such by a depth-first search.

The weighting algorithm uses natural loops to identify loops and loop-exit edges. The natural loop of a backedge $x \rightarrow y$ is defined as follows:

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

A vertex y is a loop-entry if it is the target of one or more backedges. The natural loop of a loop-entry y, denoted nat-loop(y), is simply the union of all the natural loops nat-loop($x \rightarrow y$), where $x \rightarrow y$ is a backedge.

If a and b are different loop-entry vertices, then either nat-loop(a) and nat-loop(b) are disjoint, or one is entirely contained within the other. This nesting property is used in the definition of the exit edges of a loop with loop-entry y:

$$\text{exit-edges}(y) = \{\, a \rightarrow b \mid a \rightarrow b \in E, a \in \text{nat-loop}(y), b \notin \text{nat-loop}(y), \text{ and there is no loop-entry } z\ (z \neq y) \text{ such that } a \in \text{nat-loop}(z) \text{ and } \text{nat-loop}(z) \subseteq \text{nat-loop}(y) \,\}$$

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

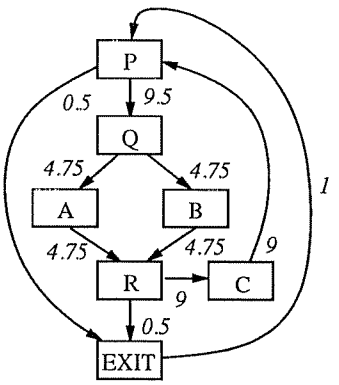
The weighting algorithm assumes that each loop iterates LOOP_MULTIPLIER times (for our implementation, set to 10) and that each branch of a predicate is equally likely to be chosen. Exit edges are specially handled, as described below. Initially, the weight of each edge and vertex is 0. The weight of the edge $EXIT \rightarrow root$ is fixed at 1 and does not change. The weighting algorithm simply iterates the following rules until they converge (*i.e.*, the application of any of the rules will not change any weight):

(1) If vertex v is a loop-entry with weight W and $N = |\text{exit-edges}(v)|$, then each edge in exit-edges(v) gets weight W/N.

(2) If vertex v has weight W and W_{EXIT} is the sum of the weights of the outgoing edges of v that are exit-edges, then each non-exit outgoing edge of v gets the weight $(W - W_{EXIT})/N$, where N is the number of non-exit outgoing edges of v. If v is a loop-entry, then each non-exit outgoing edge of v gets the weight (LOOP_MULTIPLIER * $W - W_{EXIT}$)/N.

(3) The weight of a vertex is the sum of the weights of its incoming (non-backedge) edges.

The following example illustrates the above definitions and shows the weighting that the algorithm will compute for the CFG from Figure 1:



nat–loop(P) = nat–loop(C–>P) ={ **P, Q, A, B, R, C** }

exit–edges(P) = { **P–>EXIT, R–>EXIT** }

REFERENCES

1. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, MA (1986).

2. 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 (published as SIGARCH Computer Architecture News)* **19**(2) pp. 290-302 (April 1991).

3. Systems Performance Evaluation Cooperative, *SPEC Newsletter (K. Mendoza, editor)* **1**(1)(1989).

4. 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 SIPLAN 1984 Symposium on Compiler Construction (SIPLAN Notices)* **19**(6) pp. 37-47 (June 1984).

5. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman, San Francisco (1979).

6. 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).

7. J. L. Kennington and R. V. Helgason, *Algorithms for Network Programming,* Wiley-Interscience, John Wiley and Sons, New York (1980).

8. D. E. Knuth and F. R. Stevenson, "Optimal Measurement Points for Program Frequency Counts," *BIT* **13** pp. 313-322 (1973).

9. J. R. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs," *Software Practice and Experience* **20**(12) pp. 1241-1258 (December, 1990).

10. 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).

11. 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).

12. B. P. Miller and J. D. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *Proc. of the ACM SIPLAN 1988 Conf. on Prog. Lang. Design and Implementation (SIPLAN Notices)* **23**(7) pp. 135-144 (June 1988).

13. 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).

14. K. Pettis and R. C. Hanson, "Profile Guided Code Positioning," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (published as SIGPLAN Notices)* **25**(6) pp. 16-27 ACM, (June, 1990).

15. 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).

16. 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).

17. V. Sarkar, "Determining Average Program Execution Times and their Variance," *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (published as*

SIGPLAN Notices) **24**(7) pp. 298-312 ACM, (June 21-23, 1989).

18. A. J. Smith, "Cache Memories," *ACM Computing Surveys* **14**(3) pp. 473-530 (1982).
19. MIPS Computer Systems, Inc., *UMIPS-V Reference Manual (pixie and pixstats),* MIPS Computer Systems, Sunnyvale, CA (1990).
20. R. E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, PA (1983).