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

ABSTRACT

This paper presents algorithms for inserting monitoring code to profile and trace programs. These algorithms greatly reduce the cost of measuring programs. Profiling counts the number of times each basic block in a program executes and has a variety of applications. Instruction traces are the basis for trace-driven simulation and analysis, and are also used in trace-driven debugging.

The profiling algorithm 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. 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. The profiling algorithm reduces the number of counters by a factor of two and the number of counter increments by up to a factor of four. The tracing algorithm reduces the file size and overhead of an already highly optimized tracing system by 20-40%.

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

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 [2,4,5,10,12,13,18]. Instruction traces are the basis for trace-driven simulation and analysis and are also used in trace-driven debugging [8,11,19].

Our goal is an *exact* basic block profile or trace—as opposed to the Unix *prof* command, which samples the program counter during program execution. This paper shows how to significantly reduce the cost of exact 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 an entire trace can be efficiently regenerated.

Both algorithms have been implemented and substantially improve performance over previous approaches.

Each algorithm 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. The algorithms for profiling and tracing programs are based on the well-known maximum spanning tree algorithm, applied to the program's control-flow graph [21].

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 between blocks, instrumentation code can be placed on vertices, edges, or some combination of the two. This work shows that for both profiling and tracing, it is 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. Weightings can be obtained either by empirical measurement (*i.e.*, profiling) or a heuristic estimation. Our results show that a simple edge frequency heuristic is accurate in predicting areas of low execution frequency at which to place instrumentation code.

The 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. 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, they 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.

2. BACKGROUND

A control-flow graph (CFG) is a rooted directed graph $G = (V, E)$ with a 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* the 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 Kirchoff's law of conservation of flow: for each vertex v, the sum of the weights of edges with target v (the *incoming* edges of v) must be equal to the sum of the weights of edges with source v (the *outgoing* edges of v). The weight of a vertex is simply the sum of the weights of its incoming (or outgoing) edges. If W is a weighting of CFG G, then for a set of edges and vertices pl from CFG G, cost(G, pl, W) is the sum of the weights on the edges and vertices in pl.

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 the sequence of vertices from such a directed path). 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 $EXIT \rightarrow root$ is defined to be 1.

The edge frequencies for any execution (or set of executions) of a CFG constitute a weighting of the CFG. Conversely, a heuristically chosen weighting can summarize many different executions.

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. Maximum spanning trees can be computed efficiently by a variety of algorithms [21].

Figure 1 illustrates these definitions. The first graph is the CFG of the program shown—this graph has been given a weighting. The second graph is a maximum spanning tree of the first graph. 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 $C \rightarrow P \leftarrow EXIT$.

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 (pixie and other instrumentation tools use this method [20]). 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.

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

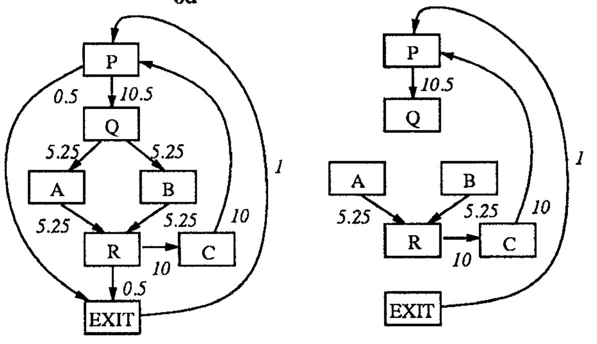

Figure 1. A program, its CFG with a weighting, and a maximum spanning tree. The edge *EXIT*→*P* is needed so that the flow equations for the root vertex (*P*) and *EXIT* are consistent. This edge does not correspond to an actual flow of control and is not instrumented.

(a)
$$\begin{tikzcd} EFreq(G,epl) \arrow[dr, "\leq"] & & VFreq(G,vpl) \arrow[dl, "\leq"'] \\ & VFreq(G,epl) & \end{tikzcd}$$

(b)
$$EFreq(G,epl) = VFreq(G,epl) \leq VFreq(G,vpl)$$

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.

Let *pl* be a subset of the edges and/or vertices from CFG *G*. The set *pl* solves the *vertex frequency* problem for CFG *G*, denoted *VFreq*(*G*,*pl*), iff the frequency of each vertex in any execution of *G* can be deduced solely from the CFG *G* and the frequencies of the edges and vertices in *pl*. The set *pl* contains those edges and vertices whose frequencies are directly measured by counters. To reduce the cost of profiling (*i.e.*, the number of counter increments), these counters should be placed in areas of low execution frequency. That is, *pl* should solve *VFreq*(*G*,*pl*) and minimize cost(*G*, *pl*, *W*) for a weighting *W*. Such a *pl* is referred to as an optimal solution to *VFreq*(*G*,*pl*) (with respect to weighting *W*).

Similarly, a set *pl* solves the *edge frequency* problem for CFG *G*, denoted *EFreq*(*G*,*pl*), iff the frequency of each edge in any execution of *G* can be deduced solely from the CFG *G* and the frequencies of the edges and vertices in *pl*. 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 *VFreq*(*G*,*epl*), *VFreq*(*G*,*vpl*), and *EFreq*(*G*,*epl*), with the goal of optimally solving *VFreq*(*G*,*pl*). Since there are CFGs for which there are no *vpl* solutions to the edge frequency problem, *EFreq*(*G*,*vpl*) is not considered [15]. This section presents three results:

(1) an algorithm to optimally solve *EFreq*(*G*,*epl*)

(2) a comparison of the optimal solutions to *VFreq*(*G*,*epl*), *VFreq*(*G*,*vpl*), and *EFreq*(*G*,*epl*). Case (a) of Figure 2 summarizes the relationship between these three problems

(3) a proof that an optimal solution to *EFreq*(*G*,*epl*) is also an optimal solution to *VFreq*(*G*,*epl*) for a large class of structured CFGs

3.1. The Edge-Frequency/Edge-Placement Problem

To solve *EFreq*(*G*,*epl*), 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 *EFreq*(*G*,*epl*) *iff* (*E*−*epl*) contains no (possibly undirected) cycle [6]. 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 *EFreq*(*G*,*epl*) iff *E*−*epl* is a spanning tree of *G*. Thus, the minimum number of counters necessary to solve *EFreq*(*G*,*epl*) is $|E| - (|V| - 1)$.

To see how such a placement solves the edge frequency problem, consider a CFG *G* and a set *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 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*−*epl* are uniquely determined. If *E*−*epl* is not a spanning tree of *G* (*i.e.*, there is a cycle, possibly undirected, in *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.

Any of the well-known maximum spanning tree algorithms described by Tarjan [21] will produce the maximum spanning tree of G with respect to weighting W. The edges that are not in the spanning tree solve $EFreq(G,epl)$ and minimize cost(G, *epl*, W).

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.

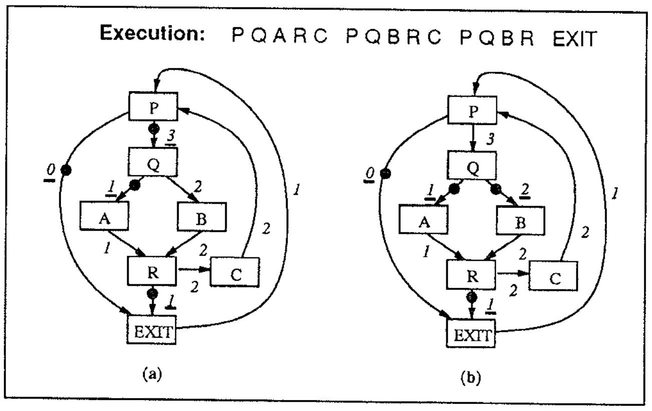

Figure 3. Solving $EFreq(G,epl)$ using the spanning tree. The dotted edges are in *epl* and the remaining edges ($E-epl$) form a spanning tree of the CFG. The frequency of each edge in the execution is shown and the measured frequencies are underlined. For the weighting given in Figure 1, the *epl* in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

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) = 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) = 42. For this example, the *epl* placement of case (a) is suboptimal for any weighting.

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 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 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 the number of times the procedure G executed.

(2) The simple extension for 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 can be handled in our framework. However, dynamically-computed interprocedural jumps (*e.g.*, *setjmp*/*longjmp*) can cause problems. The common case of the call to the exit procedure that terminates execution illustrates this problem. In this case, the information on the activation stack at program termination is sufficient to correct the count error. Figure 4 describes this problem and a solution.

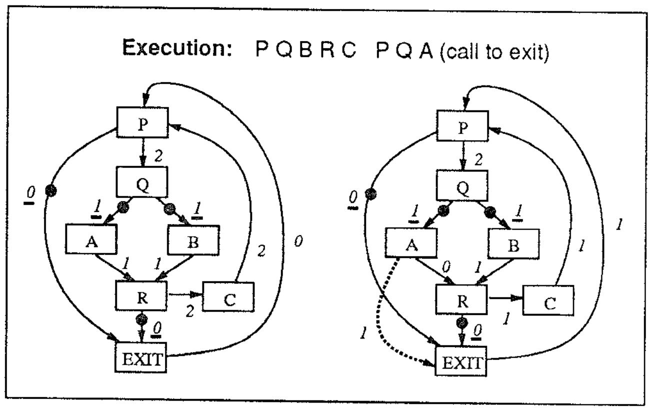

Figure 4. The first CFG executes the path shown and calls the exit routine at vertex A, which terminates the program. If the measured counts (underlined) are propagated to the spanning tree edges, incorrect values are computed. The second graph shows how this problem is solved. At program termination, 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. One such edge must be added for each active procedure on the stack at program termination.

3.2. Comparing the Three Frequency Problems

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

We first consider why *VFreq(G,epl)*, *VFreq(G,vpl)*, and *EFreq(G,epl)* are the most interesting problems to study. Suppose that a set *pl* contains a mix of vertices and edges and optimally solves *VFreq(G,pl)* or *EFreq(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*, cost(*G*, *pl'*, *W*) $\leq$ cost(*G*, *pl*, *W*).[1] 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*, an optimal solution to *VFreq(G,vpl)* is never better than an optimal solution to *VFreq(G,epl)*. An example where *EFreq(G,epl)* (and thus *VFreq(G,epl)*) betters *VFreq(G,vpl)* is discussed later.

Since any solution to *EFreq(G,epl)* must also solve *VFreq(G,epl)*, it is clear that an optimal solution to *EFreq(G,epl)* can never be better than an optimal solution to *VFreq(G,epl)* for a given CFG and weighting. As Figure 5 illustrates, there are cases where an optimal solution to *VFreq(G,epl)* is better than an optimal solution to *EFreq(G,epl)*. The only examples that we have encountered in which *VFreq(G,epl)* betters *EFreq(G,epl)* exhibit unstructured control-flow such as found in Figure 5. For the CFG in Figure 1, the optimal solution to *EFreq(G,epl)* is also an optimal solution to *VFreq(G,epl)*. Section 3.3 describes a class of graphs for which an optimal solution to *EFreq(G,epl)* is an optimal solution to *VFreq(G,epl)*.

Finally, in comparing *EFreq(G,epl)* and *VFreq(G,vpl)* (for general CFGs), there are examples in which one is better than the other and vice versa. Case(b) of Figure 5 can be easily modified to show an example where *VFreq(G,vpl)* is better than *EFreq(G,epl)*: simply consider

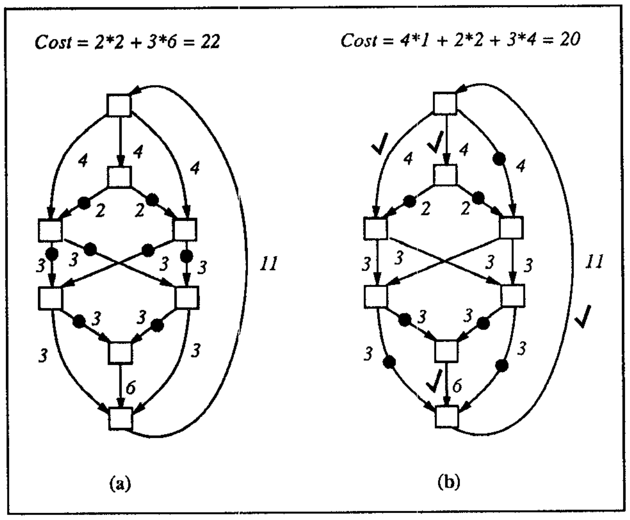

Figure 5. An example of a CFG and a weighting for which an optimal solution to *VFreq(G,epl)* is better than an optimal solution to *EFreq(G,epl)*. The dotted edges are in *epl*. Case (a) shows an optimal solution to *EFreq(G,epl)*. The edges in *E−epl* form a maximum spanning tree of the graph. The lower cost *epl* placement in case (b) does not solve *EFreq(G,epl)* (as there is a cycle in *E−epl*) but does solve *VFreq(G,epl)*. To see this, note that the count for each checked edge is uniquely determined by the counts for the dotted edges and that this yields enough edge counts to determine the count for every vertex. The counts of the four edges in the inner cycle are not uniquely determined.

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 *VFreq(G,vpl)* with cost 20. The optimal solution to *EFreq(G,epl)* for this graph still has cost 22.

There are many examples of structured CFGs where *EFreq(G,epl)* is preferable to *VFreq(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 *VFreq(G,vpl)* is { *A*, *B*, *C*, *EXIT* }, with a cost of 21.5. The optimal solution to *EFreq(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 *VFreq(G,pl)* is to optimally solve *VFreq(G,epl)*. Unfortunately, *VFreq(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

[1]Placing counters along edges instead of on vertices may require insertion of jumps in addition to counting code, which is not reflected in our cost metric. See Samples for an excellent discussion of the problem of optimizing jumps in addition to counting code [17].

for most CFGs encountered in practice, an optimal solution to *EFreq(G,epl)* will provide an optimal (or near-optimal) solution to *VFreq(G,epl)*. This section describes a class of CFGs for which an optimal solution to *EFreq(G,epl)* is also an optimal solution to *VFreq(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 p and z are the only vertices common to both PTH_a and PTH_b.

THEOREM 3.1. If *epl* solves *VFreq(G,epl)*, then $E-epl$ contains no diamond or directed cycle.

PROOF. If $E-epl$ contains a diamond or a directed cycle, then it is possible to find two executions of G such that the frequency of each edge in *epl* is the same in both executions, and where there is a vertex v with a different frequency in each execution. Since *epl* does not distinguish these two different executions, *epl* cannot solve *VFreq(G,epl)*. □

COROLLARY 3.2. For any CFG G with weighting W, an optimal *epl* solution to *VFreq(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 *VFreq(G,epl)* can never be better than an optimal solution to *EFreq(G,epl)*. Therefore, for this class of CFGs, an optimal solution to *EFreq(G,epl)* is an optimal solution to *VFreq(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. The largest subset of structured CFGs contained in G^* are those CFGs generated by **while** loops, **if-then-else** conditionals, and **begin-end** blocks. However, in general, CFGs generated by programs with **repeat** loops or breaks are not always members of G^*.

To date, we have not found any examples of CFGs generated by structured programs with multi-exit loops for which *VFreq(G,epl)* betters *EFreq(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. The *tracing problem* is to record enough information about a program's execution to be able 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 [8].

Assuming that there is 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 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. Section 4.2 considers the complications introduced by multi-procedure programs.

4.1. Single-Procedure Tracing

In this section, assume that basic blocks do not contain any 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 *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:

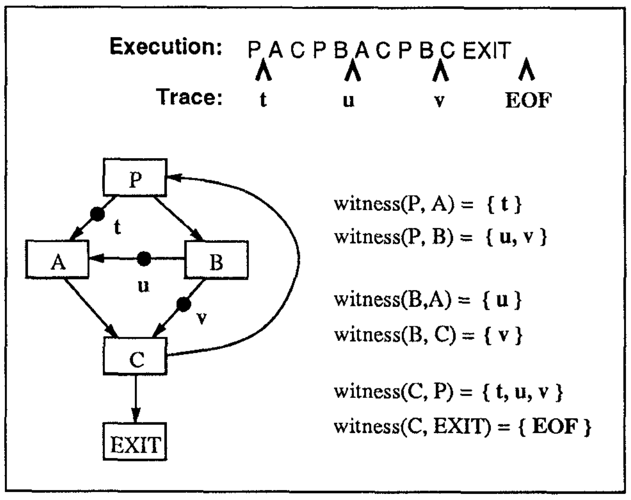

Figure 6. Example of a traced function. Vertices P, B, and C are predicates. The witnesses are shown by dots along edges. For the execution shown, the traced generated is (**t, u, v, EOF**). The witness **EOF** is always the last witness in a trace. The execution can be reconstructed from the trace using the witness sets to guide which branches to take.

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

Figure 6 illustrates the above definitions.

Let us examine how the execution in Figure 6 can be regenerated from its 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 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 from the trace. 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 $witness(G,\ epl,\ p,\ a)$ 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 6, 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, *epl*, solves the *tracing problem* for CFG G, denoted $Trace(G,epl)$, iff for each predicate p in G with successors $q_1, \ldots, q_m$, for all pairs $(q_i,\ q_j)$ such that $i \neq j$,

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

A witness placement *epl* and an execution *EX* determine a trace as follows: let trace_record(*EX*, *epl*) = $(w_1, \ldots, w_k) \parallel EOF$, 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 $Trace(G,epl)$, and trace_record(*EX*, *epl*), the algorithm in Figure 7 regenerates the execution *EX*. The following theorem captures the correctness of this algorithm:

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

PROOF. Omitted. See [1] for details. □

```
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 $pc \rightarrow newpc \in epl$ then
                wit := read(trace)
            fi
        else
            if wit = NULL then wit := read(trace) fi
            newpc := q such that
                    $wit \in witness(G,\ epl,\ pc,\ q)$
        fi
        if $pc \rightarrow newpc \in epl$ then wit := NULL fi
        pc := newpc;
        output(pc);
    until ( pc = EXIT )
end
```

Figure 7. Algorithm for regenerating an execution from a trace.

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

PROOF. Omitted. See [1] for details. □

Solving *Trace*(*G*,*epl*) so that cost(*G*,*epl*,*W*) is minimized (where *W* is a weighting) is an NP-complete problem [14]. The reduction is similar to that used to show the Uniconnected Subgraph problem is NP-complete [9], but is complicated by the fact that a weighting satisfies Kirchoff's flow law. However, for any CFG *G* in G^*, an optimal solution to *EFreq*(*G*,*epl*) is an optimal solution to *Trace*(*G*,*epl*).

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 6. Suppose that basic block *B* contains a call to procedure *X* and executions proceeds from *P* 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 instrumentation and is complicated by multiple calls to the same 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.

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 witness is guaranteed to have been generated by an edge in *G*.[2]

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 the *EXIT* vertex or a vertex containing a call.

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 *Trace*(G_i, epl_i) and epl_i has the blocking property for G_i.

The regeneration algorithm in Figure 7 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 continues executing the caller from the point of the call.

An easy way to ensure that *epl* has the blocking property is to include each incoming edge to a call or *EXIT* vertex in *epl*. Figure 8 illustrates the reasons why this approach is suboptimal. 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. 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*(*G*,*epl*).

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

[2]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.

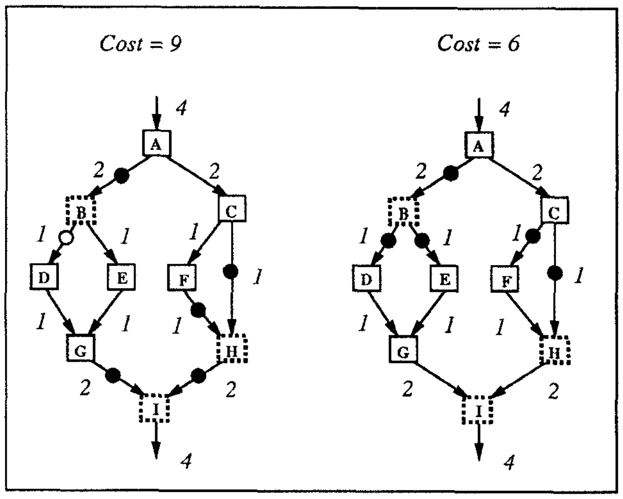

Figure 8. Two placements of blocking witnesses. The dashed vertices (B, I, and H) are call vertices. In the first subgraph, a blocking witness is placed on each incoming edge to a call vertex (black dots). This placement is suboptimal because the witness on edge $H \rightarrow I$ is not needed and because 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.

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

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

The first step of the algorithm adds an edge e to (initially empty) *epl* whenever e is a member of blockers(G, v) and v is a call or *EXIT* vertex. To ensure that *epl* solves *Trace*(G,*epl*), edges are then added to *epl* so that $E-epl$ contains no diamonds or directed cycles. The maximum spanning tree algorithm, modified so that no edge already in *epl* is allowed in the spanning tree, is applied to G. The edges that are not in the spanning tree are added to *epl*, which guarantees that *epl* solves *Trace*(G,*epl*).[3]

5. PERFORMANCE

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

[3]The modified spanning tree algorithm may not actually be able to create a spanning tree of 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.

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* [20]. 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). The algorithm uses a heuristic weighting, based on the assumptions that (1) each loop iterates ten times, (2) if a loop is entered N times and has E exit edges then each exit edge gets weight N/E, and (3) predicates are equally likely to take any of their non-exit branches (see [1] for details).

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 and 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. As can be seen from the "Slow" and "Quick" columns, 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. The "Feedback" column shows that the heuristic weighting is good at identifying regions of low execution frequency. The times for *pixie* 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 profiling using this 5 instruction code sequence.

Table 2 shows the 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

SPEC Benchmark	Slow (sec.)	Slow %	Quick (sec.)	Quick %	Feedback (sec.)	Feedback %	Pixie (sec.)	Pixie %	Quick+ (sec.)	Quick+ %
gcc (C)	32.2	222.0	19.5	95.0	16.1	61.0	24.5	145.0	14.3	43.2
espresso (C)	71.5	177.1	45.6	76.7	41.0	58.9	52.6	103.9	34.8	34.9
spice	379.9	62.7	320.7	37.3	327.3	40.2	320.8	37.4	273.1	17.0
doduc	197.5	56.6	142.6	13.1	136.7	8.4	180.1	42.8	133.6	5.9
nasa7	1,045.9	15.7	1,025.9	13.5	1023.5	13.2	992.4	9.8	959.3	6.1
li (C)	945.9	218.9	553.6	86.6	498.4	68.0	808.2	172.5	413.4	39.4
eqntott (C)	313.1	423.6	122.5	104.8	121.6	103.3	178.7	198.8	88.3	47.7
matrix300	311.5	13.6	308.8	12.6	311.0	13.4	292.4	6.6	290.0	5.7
fpppp	240.2	36.2	199.7	13.2	198.8	12.7	207.2	17.5	187.0	6.0
tomcatv	179.4	10.7	176.6	8.9	172.9	6.7	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. The column labeled Feedback shows the profiling overhead when the algorithm used an exact weighting from a previous run with identical input. 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.

SPEC Benchmark	Counter Increments: Slow	Quick	Slow/ Quick	Feedback	Slow/ Feedback	Dynamic Block Size
gcc (C)	27,149,754	8,458,003	3.2	5,324,315	5.1	4.6
espresso (C)	91,259,523	33,139,589	2.8	27,247,737	3.3	5.0
spice	308,194,784	180,543,666	1.7	172,595,830	1.8	10.6
doduc	130,897,009	45,651,338	2.9	35,920,460	3.6	11.2
nasa7	298,530,617	254,628,038	1.2	251,638,412	1.2	30.2
li (C)	1,208,747,235	413,622,801	2.9	289,473,770	4.2	4.1
eqntott (C)	465,938,460	114,410,157	4.1	112,562,938	4.1	2.3
matrix300	60,035,631	54,951,383	1.1	54,947,186	1.1	46.1
fpppp	25,932,871	6,186,762	4.2	4,098,093	6.3	100.8
tomcatv	35,012,274	27,762,776	1.3	21,254,823	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 using an exact weighting from a previous run with identical input. The last column is the average dynamic basic block size.

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 11 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 3-4 and the overhead by a factor of 2-4.

Table 2 also demonstrates that the heuristic weighting algorithm is good. As can be seen from the "Feedback" column, 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 a factor of two times 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.

5.2. Tracing Performance

The witness placement algorithm was also implemented in the AE program tracing system [8]. 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

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.

reconstruct data addresses as well as the witnesses. The combined file requires the changes to the placement algorithm described in Section 4.2.

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 new tracing algorithm. For the programs with more complex control-flow, *compress* and *polyd*, the new 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.

Edge-Frequency/Edge-Placement

The solution to *EFreq*(G,epl) has been around for quite some time. In the area of network programming, the problem is known as the specialization of the simplex method to the network program [6]. The spanning tree solution is also discussed in [7, 15], among other places.

Samples considers a refinement of *EFreq*(G,epl) where the cost function models the fact that a jump may have to be inserted into the profiled program when placing a counter on an edge [17].

Sarkar describes how to choose profiling points using control dependence and has implemented a profiling tool for the PTRAN system [18]. His algorithm finds a minimum size *epl* that solves *EFreq*(G,epl) 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 the work reported here: **(1)** The algorithm only works for a subclass of reducible CFGs; **(2)** The algorithm does not use a weighting 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.

Vertex-Frequency/Vertex-Placement

Knuth and Stevenson exactly characterize when a set of vertices *vpl* solves *VFreq*(G,vpl) and show how to compute a minimum size *vpl* that solves *VFreq*(G,vpl) [7]. They construct a graph G' from CFG G such that *vpl* solves *VFreq*(G,vpl) iff epl' solves *EFreq*(G',epl'), where *vpl* can be derived easily from epl'. The authors note that their algorithm can be modified to compute a minimum cost *vpl* solution to *VFreq*(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.

Edge-Frequency/Vertex-Placement

Probert discusses the problem of solving *EFreq*(G,vpl), which is not always possible in general [15]. Using graph grammars, he characterizes a set of "well-delimited" programs for which *EFreq*(G,vpl) can always be solved. This class of graphs arises by introducing "delimiter" vertices into well-structured programs. Probert is also concerned with finding a minimal number of vertex measurement points as opposed to a minimal cost set of measurement points.

The Tracing Problem

Ramamoorthy, Kim, and Chen consider how to instrument 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 trace multi-procedure programs.

The authors are interested in finding a minimal *size* solution to *Trace*(G,epl), an NP-complete problem [9].

However, a minimum size solution does not necessarily yield a minimum cost solution, as case (a) of Figure 3 illustrates.

7. SUMMARY AND FUTURE WORK

This paper introduced 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 by instrumenting edges? Second, are there other classes of graphs for which an optimal solution to the edge frequency problem also optimally solves the vertex frequency or tracing problem? Finally, can better weighting approximation algorithms be found?

The profiling and tracing algorithms have been implemented in a tool called QP and the tracing algorithm is part of the AE tracing system [8]. 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. Samuel Bates, Paul Adams, Phil Pfeiffer, and Peter Sweeney critiqued many descriptions of the work in progress. Thanks 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. Thanks also to Vivek Sarkar for his comments and suggestions.

REFERENCES

1. T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs," Technical Report #1031, University of Wisconsin, Madison (July 1991).
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 (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 SIGPLAN 1984 Symposium on Compiler Construction (SIGPLAN Notices)* **19**(6) pp. 37-47 (June 1984).
5. 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).
6. J. L. Kennington and R. V. Helgason, *Algorithms for Network Programming,* Wiley-Interscience, John Wiley and Sons, New York (1980).
7. D. E. Knuth and F. R. Stevenson, "Optimal Measurement Points for Program Frequency Counts," *BIT* **13** pp. 313-322 (1973).
8. J. R. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs," *Software Practice and Experience* **20**(12) pp. 1241-1258 (December, 1990).
9. 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).
10. 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).
11. B. P. Miller and J. D. Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (SIGPLAN Notices)* **23**(7) pp. 135-144 (June 1988).
12. 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).
13. 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).
14. S. Pottle, *private communication.* October 1991.
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. A. D. Samples, "Profile-Driven Compilation," Ph. D. Thesis (Report No. UCB/CSD 91/627), University of California at Berkeley (April 1991).
18. 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).
19. A. J. Smith, "Cache Memories," *ACM Computing Surveys* **14**(3) pp. 473-530 (1982).
20. MIPS Computer Systems, Inc., *UMIPS-V Reference Manual (pixie and pixstats),* MIPS Computer Systems, Sunnyvale, CA (1990).
21. R. E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, PA (1983).