Optimally Profiling and Tracing Programs

Thomas Ball (`tball@research.bell-labs.com`)
James R. Larus (`larus@cs.wisc.edu`)

July 1994

Optimally Profiling and Tracing Programs

THOMAS BALL and JAMES R. LARUS
University of Wisconsin – Madison

This paper describes algorithms for inserting monitoring code to profile and trace programs. These algorithms greatly reduce the cost of measuring programs with respect to the commonly-used technique of placing code in each basic block. Program profiling counts the number of times each basic block in a program executes. Instruction tracing records the sequence of basic blocks traversed in a program execution. The algorithms optimize the placement of counting/tracing code with respect to the expected or measured frequency of each block or edge in a program's control-flow graph. We have implemented the algorithms in a profiling/tracing tool and they substantially reduce the overhead of profiling and tracing.

We also define and study the hierarchy of profiling problems. These problems have two dimensions: what is profiled (*i.e.*, vertices (basic blocks) or edges in a control-flow graph) and where the instrumentation code is placed (in blocks or along edges). We compare the optimal solutions to the profiling problems and describe a new profiling problem: basic block profiling with edge counters. This problem is important because an optimal solution to any other profiling problem (for a given control-flow graph) is never better than an optimal solution to this problem. Unfortunately, finding an optimal placement of edge counters for vertex profiling appears to be a hard problem in general. However, our work shows that edge profiling with edge counters works well in practice because it is simple and efficient and finds optimal counter placements in most cases. Furthermore, it yields more information than a vertex profile. Tracing also benefits from placing instrumentation code along edges rather on vertices.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement Techniques; D.2.2 [**Software Engineering**]: Tools and Techniques—programmer workbench; D.2.5 [**Software Engineering**]: Testing and Debugging—*diagnostics, tracing*

General Terms: Algorithms, Measurement

Additional Key Words and Phrases: Profiling, instruction tracing, instrumentation, control-flow graph

---

1. INTRODUCTION

A well-known technique for recording program behavior and measuring program performance is to insert code into a program and execute the modified program. This paper discusses how to insert monitoring code to either profile or trace programs. Program profiling counts the number of times that each basic block or control-flow edge in a program executes. It is widely used to measure instruction set utilization, identify program bottlenecks, and estimate program execution times for code optimization [5, 7, 12, 21-23, 29]. Instruction tracing records the sequence of basic blocks traversed in a program execution. It is the basis for trace-driven architectural simulation and analysis and is also used in trace-driven debugging [4, 17, 30]. Both techniques have been implemented in a wide variety of systems.

In this paper, we describe algorithms for placing profiling and tracing code that greatly reduce the cost of measuring programs, compared to previously implemented approaches. The algorithms reduce measurement overhead in two ways: by inserting less instrumentation code and by placing the code where it is less likely to be executed. The algorithms have been implemented in a widely-distributed profiling/tracing tool called *qpt* [18], which instruments executable files, and performs very well in practice.

As described in Section 7, there has been considerable work on efficiently profiling and tracing programs. Three factors significantly distinguish our work from previous work. First, we consider the theoretic and algorithmic underpinnings of program profiling and tracing. Second, unlike most previous work, we implemented the algorithms and experimented with different instrumentation strategies on a collection of real programs. This experience exposed deficiencies in previous algorithms and led to extensions that make these algorithms robust enough for practical use. Third, we implemented and compared several strategies for profiling and tracing. These approaches can be categorized as to whether they measure basic block or control-flow edge frequency, and whether they place instrumentation code in basic blocks or along control-flow edges. This categorization helps to relate the efficiency of various approaches. Through this categorization, we identified a new problem that has not been previously considered: basic block profiling with edge counters. This paper characterizes this new problem and compares it to existing approaches.

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

The major contributions of this paper are:

- We enumerate the space of profiling problems based on what is profiled and where profiling code is placed. A *vertex profile* counts the number of executions of each vertex (basic block) in a control-flow graph. An *edge profile* counts the number of times each control-flow edge executes. An edge profile determines a vertex profile, but the converse does not always hold. Knuth has published efficient algorithms for finding the minimum number of vertex counters necessary and sufficient for vertex profiling [16], denoted by *Vprof* (*Vcnt*), and the minimum number of edge counters for edge profiling [15], denoted by *Eprof* (*Ecnt*). We consider the new problem of finding a set of edge counters for vertex profiling, *Vprof* (*Ecnt*), and characterize when a set of instrumented edges is necessary and sufficient for vertex profiling.

- We relate the optimal solutions to three profiling problems, *Vprof* (*Vcnt*), *Eprof* (*Ecnt*), and *Vprof* (*Ecnt*), and compare their run-time overhead in practice. We show that for a given CFG and weighting, an optimal solution to *Vprof* (*Vcnt*) or *Eprof* (*Ecnt*) is never better than an optimal

solution to *Vprof* (*Ecnt*). Unfortunately, finding an optimal solution to *Vprof* (*Ecnt*) seems to be a hard problem in general. We believe the problem is NP-complete but do not have a proof as of yet. However, we show that for a large class of structured control-flow graphs, an optimal solution to *Eprof* (*Ecnt*) is an optimal solution to *Vprof* (*Ecnt*). Furthermore, we show that *Eprof* (*Ecnt*) has lower overhead than *Vprof* (*Vcnt*) in practice.

- We show that for both profiling and tracing, placing instrumentation code on edges is better than placing it on vertices. Intuitively this is because there are more edges than vertices in the control-flow graph. Instrumenting edges provides more opportunities to place instrumentation code in areas of low execution frequency.

- We give a simple heuristic for estimating execution frequencies (based on analysis of the control-flow graph) that accurately predicts areas of low execution frequency at which to place instrumentation code.

- We show that any solution to a profiling problem is sufficient to solve the tracing problem. However, such a solution is not necessarily optimal. Ramamoorthy, Kim, and Chen have given a necessary and sufficient condition for when a set of edges solves the tracing problem for single procedure programs [26]. However, this condition does not work for multi-procedure programs. We reformulate this condition in a more intuitive manner and show how it can be extended to apply to multi-procedure programs.

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

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

The remainder of this paper is organized as follows. Section 2 provides background material on control-flow graphs, weightings, and spanning trees. Section 3 shows how to profile programs efficiently and Section 4 describes how to trace programs efficiently. Section 5 presents our heuristic weighting algorithm. Section 6 presents performance results. Section 7 reviews related work on profiling, tracing, and heuristics for minimizing instrumentation overhead and estimating execution frequency. Section 8 concludes the paper.

2. BACKGROUND

This paper presents algorithms for instrumenting programs to record information about their execution-time behavior. These algorithms use the intraprocedural control-flow structure of programs in order to determine where to place instrumentation code. The programs under consideration are assumed to have been written in an imperative language with procedures, in which control-flow within a procedure is statically determinable. Interprocedural control-flow occurs mainly by procedure call and procedure return, although we will show how the algorithms can be extended to handle exceptions and interprocedural jumps. Whether or not procedures are first-class objects does not affect the instrumentation algorithms.

The algorithms require only that a control-flow graph can be constructed for each procedure in the program. It is not necessary to know which procedure is called at a particular call site.

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

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

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

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

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

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

A *spanning tree* of a directed graph $G = (V, E)$ is a subgraph $H = (V, T)$, where $T \subseteq E$, such that every pair of vertices in V is connected by a unique path (*i.e.*, H connects all vertices in V and there are no cycles in H). A *maximum* spanning tree of a weighted graph is one such that the cost of the tree edges is maximal. The maximum spanning tree for a graph can be computed efficiently by a variety of algorithms [32].

Figure 1 illustrates these definitions. The first graph is the CFG of the program shown in the figure. This graph has been given a weighting. The second graph is a maximum spanning tree of the first graph. Note that any vertex in a spanning tree can serve as a root and that the direction of the edges in the tree is
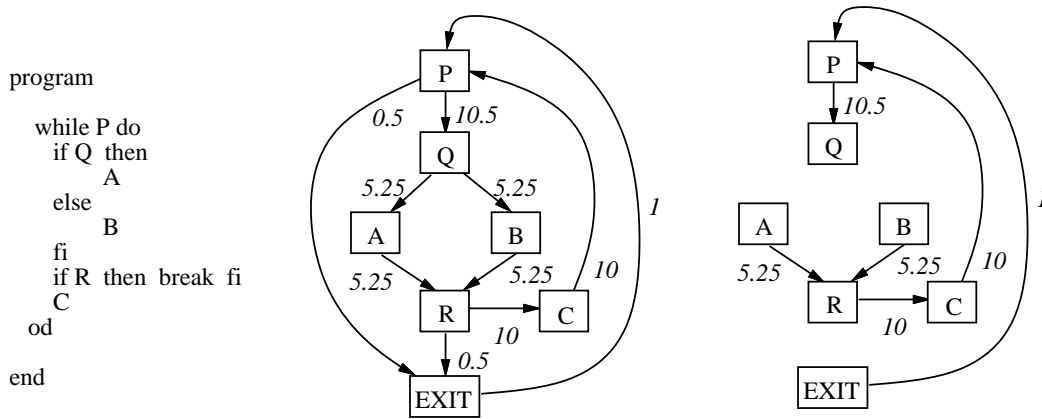
Figure 1. A program, its CFG with a weighting, and a maximum spanning tree. The edge *EXIT→P* is needed so that the flow equations for the root vertex (*P*) and *EXIT* are consistent. This edge does not correspond to an actual flow of control and is not instrumented.
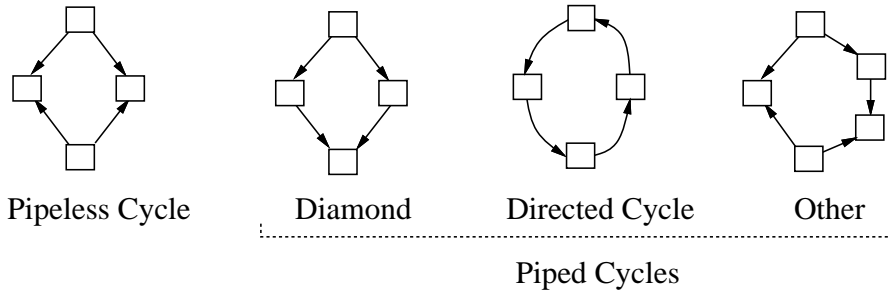
Figure 2. Classification of cycles.

unimportant. For example, vertices *C* and *EXIT* are connected in the spanning tree by the path *C→P←EXIT*.

An underlying concept in the instrumentation problems we consider is that certain cycles in a CFG must contain instrumentation code (*i.e.*, the instrumentation code must *break* certain cycles). We classify cycles based on the direction of their edges. Let u, v and w be three consecutive vertices in a cycle. There is a *fork* at *v* if $u \leftarrow v \rightarrow w$, a *join* if $u \rightarrow v \leftarrow w$, and a *pipe* otherwise ($u \rightarrow v \rightarrow w$ or $u \leftarrow v \leftarrow w$). A cycle is *pipeless* if it contains no pipes (*i.e*, the direction of edges strictly alternate around the cycle). A cycle is *piped* if it contains at least one pipe. Piped cycles are further classified: a *directed cycle* contains only pipes (all edges are in the same direction); a *diamond* is a cycle with more than two distinct edges that has exactly one fork and one join (there are two changes of direction in the cycle); *other* cycles are all other piped cycles. Figure 2 gives examples of these cycles.

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 [31]). There are two drawbacks to such an approach: (1) too many counters are used and (2) the total number of increments during an execution is larger than necessary.

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

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

Given that we can place counters on vertices or edges, a counter placement can take one of three forms: a set of edges (*Ecnt*); a set of vertices (*Vcnt*); a mixture of edges and vertices (*Mcnt*). Combined with the two profiling problems, this yields six possibilities. We do not consider *Eprof* (*Vcnt*), since there are CFGs for which there are no solutions to this problem [25]. That is, it is not always possible to determine edge frequencies from vertex frequencies. Mixed placements are of interest because placing counters on vertices rather than edges eliminates the need to insert unconditional jumps.[1] On the other hand, a vertex is executed more frequently than any of its outgoing edges, implying that it might be worthwhile to instrument some outgoing edges rather than the vertex. The usefulness of mixed placements depends on the cost of an unconditional jump relative to the cost of incrementing a counter in memory. On RISC machines (for which we constructed a profiling tool) the code sequence for incrementing a counter or generating a tracing token ranges from 5 to 11 instructions (cycles). The cost of an unconditional branch is quite small in comparison (usually 1 cycle, as the delay slot of an unconditional branch can almost always be filled with a useful instruction). In this case, there is questionable benefit from mixed placements. In fact, Samples has shown that mixed placements provide little benefit over edge placements on a machine in which the increment and branch costs were comparable, and were worse in some cases [28]. Furthermore, as shown in Section 6.1, for all the benchmarks we examined, less than half of the instrumented edges (which is about one quarter of the total number of control-flow edges) required unconditional jumps when profiling with edge counters. For these reasons, we do not consider mixed counter placements.

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

(1) A comparison of the optimal solutions to *Vprof* (*Vcnt*), *Eprof* (*Ecnt*), and *Vprof* (*Ecnt*). Figure 3(a) summarizes the relationship between these three problems for general CFGs. *X* ≤ *Y* means that for

---

[1]Placing instrumentation code along edges of the CFG essentially creates new basic blocks, which may require the insertion of unconditional jumps (assuming that the linearization of the original basic blocks is the same in the instrumented program as in the original program). On the other hand, placing instrumentation code in vertices simply expands the extent of the original basic blocks, and does not require insertion of jumps. It is possible to rearrange the placement of basic blocks to minimize the number of unconditional jumps needed, as discussed by Ramanath and Solomon [27]. However, our algorithms do not perform such an optimization, as they respect the original linearization.

any given CFG and weighting, an optimal solution to problem *X* has cost less than or equal to the cost of an optimal solution to problem *Y*. In general, for any weighted CFG, an optimal solution to *Vprof* (*Ecnt*) is always at least as cheap as *Eprof* (*Ecnt*) or *Vprof* (*Vcnt*).

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

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

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

### 3.1. Comparing the Three Profiling Problems

This section examines the relationships between the optimal solutions to *Vprof* (*Vcnt*), *Eprof* (*Ecnt*), and *Vprof* (*Ecnt*) for general CFGs, as summarized in Figure 3(a).

The three CFGs in Figure 4 illustrate optimal solutions to *Vprof* (*Vcnt*), *Eprof* (*Ecnt*), and *Vprof* (*Ecnt*) (for the weighting given in the first CFG). The black dots represent counters. The costs of the three counter placements are 124, 62 and 59, respectively. In each case, every counter is necessary to uniquely determine a profile and no lower cost placements will suffice. For example, if the counter on vertex *b* in case (a) were eliminated, it would be impossible to determine how many times *b* or *e* executed. In case (a), the counts for vertices *a*, *e*, *f*, and *EXIT* are not directly measured, but can be deduced from the measured vertices as follows: $e = b$; $a = f = EXIT = g + h$. In case (b), the count for each unmeasured edge is uniquely determined by the counts for the measured edges by Kirchoff's flow law (*e.g.*, $a \rightarrow f = f \rightarrow g + f \rightarrow h - e \rightarrow f$). In case (c), the count for each unmeasured edge except those in the set $\{a \rightarrow b, e \rightarrow b, e \rightarrow f, a \rightarrow f\}$ is uniquely determined by the measured edges. This yields enough information to deduce the count for each vertex.

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

$$\Downarrow \quad\quad\quad \Downarrow$$

Vprof(Ecnt)

(b)   *Eprof(Ecnt)* $\;=\;$ *Vprof(Ecnt)* $\;\leqq\;$ *Vprof(Vcnt)*

Figure 3. (a) The relationship between the costs of the optimal solutions of the three frequency problems for general CFGs. (b) The relationship when the CFGs are constructed from **while** loops, **if-then-else** conditionals, and **begin-end** blocks.
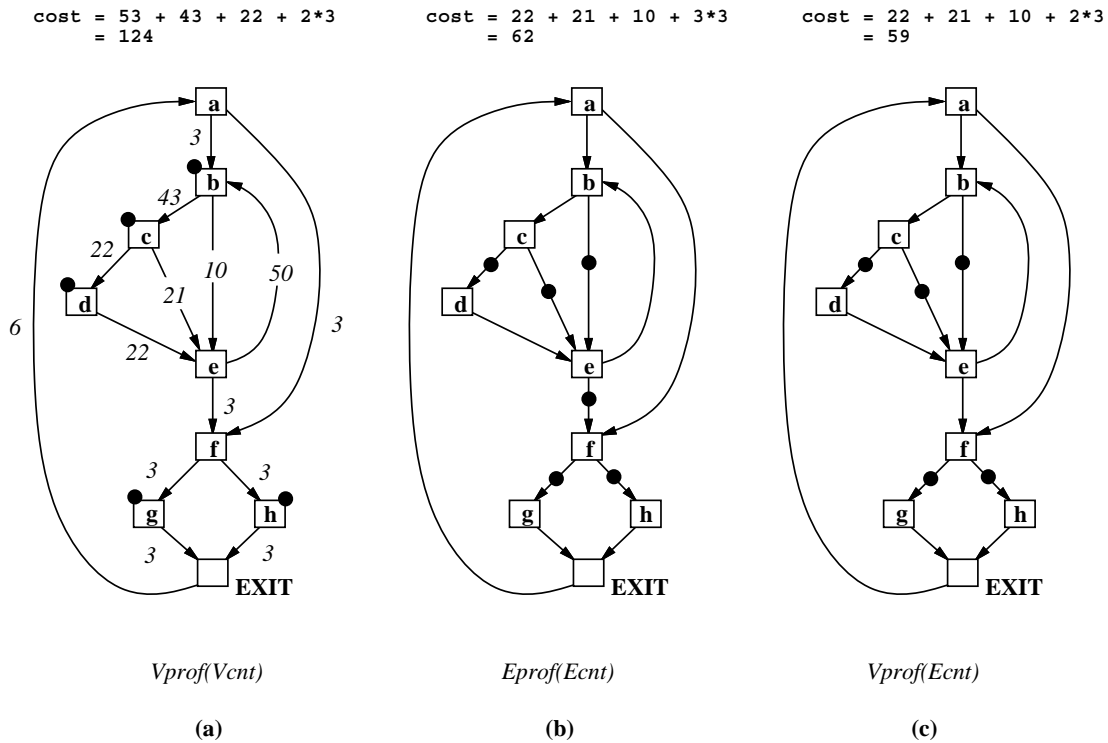
Figure 4. Optimal solutions for (a) vertex profiling with vertex counters, (b) edge profiling with edge counters and (c) vertex profiling with edge counters.

For any CFG and weighting, an optimal solution to *Vprof* (*Vcnt*) never has lower cost than an optimal solution to *Vprof* (*Ecnt*) (for every vertex *v* in *Vcnt*, *v*'s counter can be "pushed" off *v* onto each outgoing edge of *v*, resulting in counter placement *Ecnt*, which clearly solves the vertex profiling problem with cost equal to *Vcnt*). Figure 4 shows an example where *Vprof* (*Ecnt*) has lower cost than *Vprof* (*Vcnt*). The counter placement in case (c) solves *Vprof* (*Ecnt*) and has lower cost than the counter placement in case (a) that solves *Vprof* (*Vcnt*).

Since any solution to *Eprof* (*Ecnt*) must also solve *Vprof* (*Ecnt*), an optimal solution to *Eprof* (*Ecnt*) can never have lower cost than an optimal solution to *Vprof* (*Ecnt*), for a given CFG and weighting. The counter placement in case (c) solves *Vprof* (*Ecnt*) and has lower cost than the counter placement in case (b) that solves *Eprof* (*Ecnt*). In comparing *Eprof* (*Ecnt*) and *Vprof* (*Vcnt*), there are examples in which one has lower cost than the other and vice versa. Cases (b) and (a) of Figure 4 show an example where *Eprof* (*Ecnt*) has lower cost than *Vprof* (*Vcnt*). Figure 4(c) can be easily modified to show an example where *Vprof* (*Vcnt*) has lower cost than *Eprof* (*Ecnt*). Consider each black dot as a vertex in its own right and split the dotted edge into two edges. The dots constitute the set *Vcnt* and solve *Vprof* (*Vcnt*) with cost 59. The optimal solution to *Eprof* (*Ecnt*) for this graph still has cost 62.

3.2. Edge Profiling with Edge Counters

Eprof (*Ecnt*) can be solved by placing a counter on the outgoing edges of each predicate vertex. However, this placement uses more counters than necessary. Knuth describes how it follows from Kirchoff's law that an edge-counter placement *Ecnt* solves *Eprof* (*Ecnt*) for CFG $G = (V,E)$ *iff* $(E - Ecnt)$ contains no (undirected) cycle [15]. Since a spanning tree of a CFG represents a maximum subset of edges without a cycle, it follows that *Ecnt* is a minimum size solution to *Eprof* (*Ecnt*) *iff* $(E - Ecnt)$ is a spanning tree of G. Thus, the minimum number of counters necessary to solve *Eprof* (*Ecnt*) is $|E| - (|V|-1)$.

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

Any of the well-known maximum spanning tree algorithms described by Tarjan [32] will efficiently find a maximum spanning tree of a CFG with respect to a weighting. The edges that are not in the spanning tree solve *Eprof* (*Ecnt*) and minimize the cost of *Ecnt*. As a result, counters are placed in areas of lower execution frequency in the CFG. To ensure that a counter is never placed on *EXIT→root*, 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 associated with the CFG executed.

Figure 5(a) illustrates how the frequencies of edges in $E - Ecnt$ can be derived from the frequencies of edges in *Ecnt*. Black dots identify edges in *Ecnt*. The other edges are in $E - Ecnt$ and form a spanning tree of the CFG. The edge frequencies are those for the execution shown. However, we emphasize that the only edges for which frequencies will be recorded are the edges with black dots. Let vertex P be the root of the spanning tree. Vertex Q is a leaf in the spanning tree and has flow equation ($P→Q = Q→A + Q→B$). Since the frequencies for $P→Q$ and $Q→A$ are known, we can substitute them into this equation and derive the frequency for $Q→B$. Once the frequency for $Q→B$ is known, the frequency for $B→R$ can be derived from the flow equation for B, and so on. For the weighting given in Figure 1, the counter placement in Figure 5(a) has cost 16.75. However, Figure 5(b) shows a counter placement induced by the *maximum* spanning tree with resultant cost of 11.5.

The propagation algorithm in Figure 6 performs a post-order traversal of the spanning tree $E - Ecnt$ to propagate the frequencies of edges in *Ecnt* to the unprofiled edges in the spanning tree. The procedure DFS calculates the frequency of a spanning tree edge. Since the calculation is carried out post-order, once the last line in DFS(*G*, *Ecnt*, v, e) is reached, the counts of all edges incident to vertex v except e have been calculated. The flow equation for v states that the sum of v's incoming edges is equal to the sum of v's outgoing edges. One of these sums includes the count from edge e, which has been initially set to 0. The count for e is found by subtracting the minimum of the two sums from the maximum.

Execution: P Q A R C   P Q B R C   P Q B R   EXIT
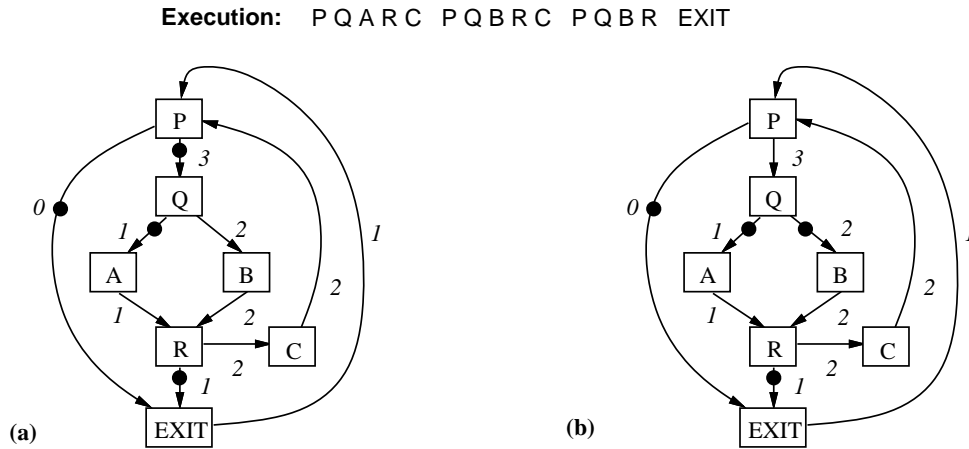
Figure 5. Solving *Eprof* (*Ecnt*) using the spanning tree. For the weighting given in Figure 1, the counter placement in case (a) is not optimal (of minimal cost) but the counter placement in case (b) is optimal.

Although profiling has been described in terms of a single CFG, the algorithm requires few changes to deal with multi-procedure programs. The pre-execution spanning tree algorithm and post-execution propagation of edge frequencies can be applied to each procedure's CFG separately. This simple extension for multi-procedure profiling will determine the correct frequencies whenever interprocedural control-flow occurs only via procedure call and return and each call eventually has a corresponding return.[2] Statically-determinable interprocedural jumps (other than procedure call and return) can be handled by adding edges corresponding to the interprocedural jumps and instrumenting these edges. Determining whether or not such an interprocedural edge needs to be instrumented would require interprocedural analysis that we do not perform.

A problem arises with dynamically computed interprocedural jumps such as setjmp/longjmp in the *C* language [14], or early program termination, as may be caused by a system call or an error condition. In these cases, one or more procedures terminate before reaching the *EXIT* vertex, breaking Kirchoff's law. For example, suppose that the CFG in Figure 7(a) executes the path shown at the top of the figure. Furthermore, suppose that the execution terminates early at vertex *A* because of a divide by zero error. As a result, control enters vertex *A* once via the edge *Q*→*A* once but never exits via *A*→*R*. However, because the propagation algorithm (see Figure 6) assumes that Kirchoff's law holds at each vertex, edge *A*→*R* will receive a count of 1, as shown in Figure 7(a). In this example, the count is off by one. However, in general, if multiple procedures on the activation stack are exited early and early exiting is a common occurrence, the counts may diverge greatly.

---

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

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

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

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

Figure 6. Edge propagation algorithm determines the frequencies of edges in the spanning tree *E-Ecnt* given the frequencies of edges in *Ecnt*. The algorithm uses a post-order traversal of the spanning tree.

Execution: P Q B R C P Q A (divide by 0)
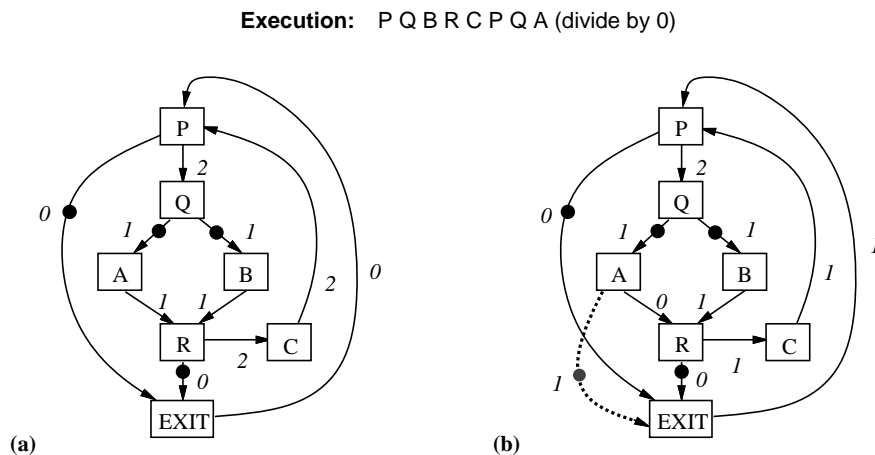
Figure 7. (a) Early termination at vertex *A* yields incorrect counts, (b) which are corrected by the addition of edge *A→EXIT*.

In this case, information available on the activation stack is sufficient to correct the count error. Conceptually, for each procedure *X* on the activation stack that exits early an edge *v→EXIT* with a count of 1 is added to procedure *X*'s CFG, where *v* is the vertex from which procedure *X* called the next procedure. This edge models early termination of procedure *X* at vertex *v*. In practice, the edge *v→EXIT* is represented by an "exit" counter that is associated with the vertex *v*. This counter is incremented once for each time procedure *X* exits early when at vertex *v*. For early termination caused by a conditional exception (such as divide by zero) the increment code must be placed in the exception handler rather than at vertex *v*, since the code should only be invoked only when *v* raises the exception. For early termination caused by longjump, the increment code must also be in the handler since longjump may pop many activation frames off the stack, each of which requires incrementing the associated exit counter.

Figure 7(b) illustrates how the early exit problem is solved. Because the procedure terminates early at vertex *A*, the edge *A→EXIT* is added to the CFG and given a count of 1. This additional edge correctly siphons off the incoming flow to vertex *A* so that the propagation algorithm yields correct counts. As shown in case(b), edge *A→R* correctly receives a count of 0.

3.3. Vertex Profiling with Edge Counters

This section addresses the problem of vertex profiling with edge counters. Section 3.3.1 characterizes when a set of edges *Ecnt* solves *Vprof* (*Ecnt*) and gives an algorithm for propagating edge frequencies through the CFG in order to determine vertex frequencies. As discussed later in Section 3.4, it appears difficult to solve *Vprof* (*Ecnt*) efficiently while minimizing the size or cost of *Ecnt*. However, as discussed in Section 3.3.2, there are certain classes of CFGs for which an optimal solution to *Eprof* (*Ecnt*) is also an optimal solution to *Vprof* (*Ecnt*). For this class of CFGs, the counter placements induced by the maximum spanning tree are optimal. Finally, Section 3.3.3 presents a heuristic for finding an *Ecnt* placement to solve *Vprof* (*Ecnt*) that improves on the spanning tree approach in certain situations.

3.3.1. Characterization and algorithm

Edge profiling with edge counters requires that every (undirected) cycle in the CFG contain a counter. Since an edge profile determines a vertex profile, vertex profiling requires no more edge counters than does edge profiling. However, as illustrated by the example in Figure 4(c), there are cases in which fewer edge counters are needed for vertex profiling than for edge profiling. In this example, there is a cycle of counter-free edges, yet there is enough information recorded to determine the frequency of every vertex. This section formalizes this observation. That is, certain types of counter-free cycles are allowed when using edge counters for vertex profiling, as captured by the following theorem:

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

Pipeless cycles are allowed in $E - Ecnt$ as well as non-simple piped cycles, as long as the simple cycles that compose them are pipeless. In Figure 4(c), the counter-free cycle represented by the set of edges { *a→b*, *e→b*, *e→f*, *a→f* } is pipeless. In Figure 9(a), the counter-free edges contain a piped cycle; however, the cycle is not simple. Both simple counter-free cycles in this example are pipeless.

Let *freq* be the function mapping edges in a CFG to their frequency in an execution. We give an algorithm that (given the frequencies of edges in *Ecnt* in the execution and the assumption that $E - Ecnt$ contains no simple piped cycle) will find a function *freq′* from edges to frequencies that is *vertex-frequency*

equivalent to *freq*. That is, for any vertex *v* the sum of the frequencies of *v*'s incoming (outgoing) edges under *freq*′ is the same as under *freq*. We first explain the algorithm and show how it operates on an example. We then prove the correctness of the algorithm, showing that if $E - Ecnt$ contains no simple piped cycle then *Ecnt* solves *Vprof* (*Ecnt*). Finally, we show that if $E - Ecnt$ contains a simple piped cycle then it is not possible for *Ecnt* to solve *Vprof* (*Ecnt*).

Figure 8 presents the propagation algorithm. The frequencies for edges in *Ecnt* have been determined by an execution *EX*. The algorithm operates as follows: while there is a (simple) cycle *C* in the set of edges $E - (Ecnt \cup Break)$, an edge *e* from cycle *C* is added to the set *Break* and the frequency of edge *e* is initialized to zero. Once $E - (Ecnt \cup Break)$ is acyclic, it follows that the frequencies of edges in $Ecnt \cup Break$ uniquely determine the frequencies of the other edges (by the spanning tree propagation algorithm, as given in Figure 6). As we will show, the vertex frequencies determined by these edge frequencies are the true vertex frequencies in the execution *EX*.

Figure 9 presents an example of how this algorithm works. The CFG in Figure 9(a) contains two simple cycles in $E - Ecnt$. As usual, edges in *Ecnt* are marked with black dots. Each of the counter-free simple cycles is clearly pipeless. These two simple cycles combine into a non-simple cycle containing a pipe, which is allowed under the structural characterization of *Vprof* (*Ecnt*). The edges in the CFG are numbered with their frequencies from some execution. The frequencies of the checked edges can be derived easily from the frequencies of the edges in *Ecnt*. From these frequencies, the count of every vertex except the grey vertex can clearly be determined. How do we derive counts for the edges in the two simple pipeless cycles in order to determine the frequency of the grey vertex? Suppose the algorithm chooses to break the two simple cycles in $E - Ecnt$ by putting the dashed edges (see Figure 9(b)) into the set *Break*, giving both frequency 0, as shown in case (b). Spanning tree propagation of edge frequencies in the set $Ecnt \cup Break$ to edges in $E - (Ecnt \cup Break)$ will assign unique frequencies to the other edges in the simple pipeless cycles, as shown in case (b). The sum of the frequencies of the incoming (outgoing) edges to the grey vertex is 2, which is the correct frequency (even though the frequencies of edges in the pipeless cycle are not the same as in the execution).
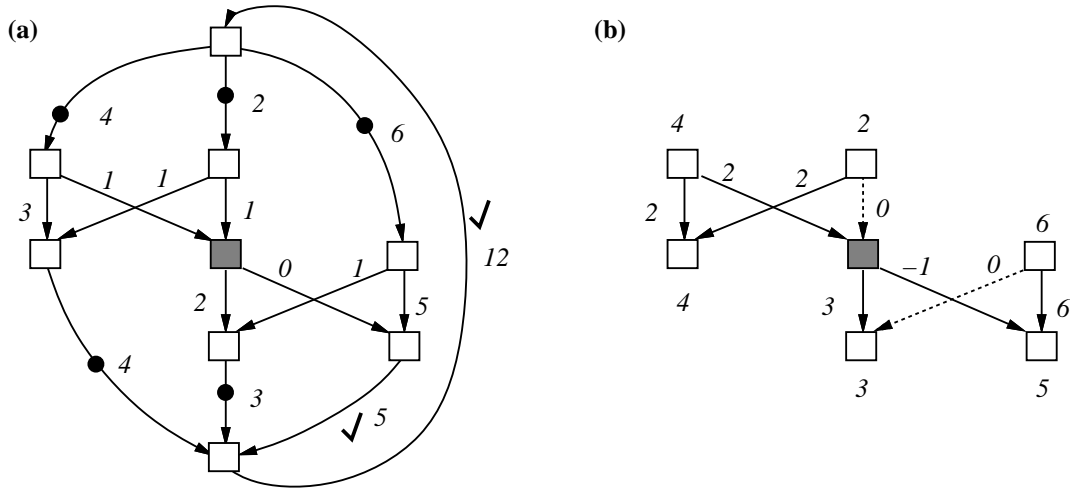

```
/* Assumption: E − Ecnt contains no simple piped cycle */
/* for each edge e in Ecnt, cnt[e] = frequency of e in execution */


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

propagate_counts(Ecnt ∪ Break)   /* from Figure 6 */
```

Figure 8. Algorithm for propagating edge counts to determine vertex counts.

Figure 9. (a) An example CFG in which *E-Ecnt* contains two simple pipeless cycles. (b) If the dashed edges are assigned frequency 0, spanning tree propagation will assign the remaining edges in the simple pipeless cycles the frequencies shown. This yields a count of two for the grey vertex, which is its correct frequency.

We now prove the correctness of the algorithm. Let *freq* be the function mapping edges in a CFG to their frequency in an execution, and let *freq* ′ be the function from edges to frequencies created by the algorithm of Figure 8. We show that *freq* ′ is vertex-frequency equivalent to *freq* by induction on the size of *Break* (as determined by the algorithm).

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

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

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

(1) Function *g* is vertex-frequency equivalent to $freq_n$;
(2) Function *g* satisfies Kirchoff's flow law at every vertex;
(3) For each edge $f \in Ecnt \cup Break$, $g(f) = freq'(f)$.

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

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

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

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

$\square$

If $E - Ecnt$ contains a simple piped cycle, then there are two executions of G with different frequencies for some vertex but for which the frequencies of edges in $Ecnt$ are the same. This is clear if $E - Ecnt$ contains a directed cycle, or two edge-disjoint directed paths between a pair of vertices (*i.e.*, a diamond). Figure 10 gives an example of a CFG in which $E - Ecnt$ contains a piped cycle (the pipe is at vertex B) that is neither a directed cycle nor a diamond and shows two different execution paths. Both execution paths traverse each instrumented edge (x, y, z) exactly once. However, EX_1 contains vertex B while EX_2 does not.

Another way to look at this is that the edge frequencies in a cycle in $E - Ecnt$ are unconstrained. Let $freq_n$ be a function mapping edges to values that satisfies Kirchoff's flow law at every vertex. Applying the function transformation defined in the above proof to $freq_n$ based on a piped cycle in $E - Ecnt$ results in
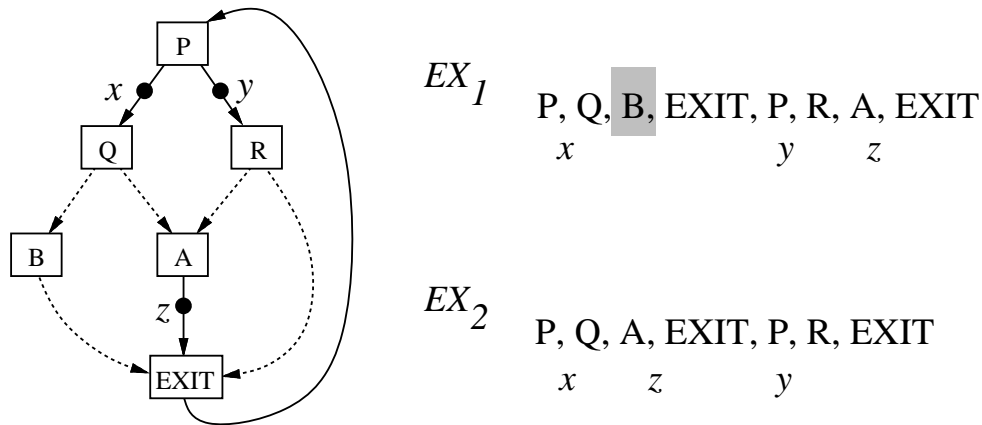
Figure 10. An example of instrumentation that is not sufficient for vertex profiling. The dashed edges in the CFG constitute a simple cycle of uninstrumented edges with a pipe (at vertex B). Executions EX_1 and EX_2 traverse each instrumented edge the same number of times but EX_1 contains B and EX_2 does not.

function *g* such that Kirchoff's flow law holds at every vertex. While the frequency of each vertex in a fork or join in the cycle remains the same (as shown above), the frequency of the vertex in the pipe will have changed.

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

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

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

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

It follows directly from these two observations that for any CFG in G^*, an optimal solution to *Eprof* (*Ecnt*) is also an optimal solution to *Vprof* (*Ecnt*). The class of graphs G^* contains CFGs with multiple exit loops (such as in Figure 1), CFGs that can only be generated using **gotos**, and even some irreducible graphs. The class G^* contains those structured CFGs generated by **while** loops, **if-then-else** conditionals, and **begin-end** blocks (because every simple cycle in these CFGs is either a directed cycle or a diamond). However, in general, CFGs of programs with **repeat-until** loops or **break**s are not always members of G^*. The CFG in Figure 4 is an example of such a graph.

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

Because we believe *Vprof* (*Ecnt*) is a hard problem to solve optimally, we developed a heuristic for *Vprof* (*Ecnt*). The heuristic first computes a maximum spanning tree *ST* (inducing a counter placement on the edges not in tree) and then checks if any counters can be removed without creating simple piped cycles in the set of counter-free edges. An algorithm for the heuristic is given in Figure 11.

The heuristic makes use of the following observation: if *ST* is a spanning tree and edge *e* is not in *ST*, then the addition of *e* to *ST* creates precisely one simple cycle C_e in *ST*. The heuristic examines each such cycle C_e in turn. To prevent two counter-free pipeless cycles from combining into a simple counter-free piped cycle, it marks all vertices in the cycle C_e when a counter is removed from *e*; a counter is removed

---

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

**Figure 11.** A heuristic for *Vprof* (*Ecnt*).

from an edge e only if cycle C_e is pipeless and contains no marked vertices. The heuristic is described in detail in Figure 11. Upon termination, the set "Remove" contains all edges whose counters can be removed safely. By considering edges in decreasing order of weight, the algorithm tries to remove counters with higher cost first.

Consider the application of the heuristic to the CFG in Figure 4. Case (b) shows the counter placement resulting from the maximum spanning tree algorithm. Removing the counter on edge $e \rightarrow f$ creates a pipeless cycle in the set of counter-free edges. Removing the counter from any other edge creates a piped cycle in the set of counter-free edges. In this example, the heuristic produces the optimal counter placement in case (c). However, there are examples for which this heuristic will not find an optimal solution to *Vprof* (*Ecnt*).

3.4. Cycle Breaking Problems

The problems of profiling and tracing programs with edge instrumentation can be described as cycle breaking problems, where certain types of cycles in the CFG must contain instrumentation code in order to solve a profiling or tracing problem. Figure 12 summarizes the classification of cycles presented in Section 2, the problems they correspond to, and the known time complexity for (optimally) breaking each class of cycle. Solving *Eprof* (*Ecnt*) corresponds to breaking all undirected cycles. Solving *Vprof* (*Ecnt*) corresponds to breaking all simple piped cycles, as we have shown in Section 3.3. Finally, as discussed in Section 4, solving the tracing problem corresponds to breaking all directed cycles and diamonds.

Of course, we are interested in a minimum cost set of edges that breaks a certain class of cycles. Finding a minimum size set of edges that breaks all directed cycles is an NP-complete problem (Feedback Arc Set [9]). Maheshwari showed that finding a minimum size set of edges that breaks diamonds is also NP-
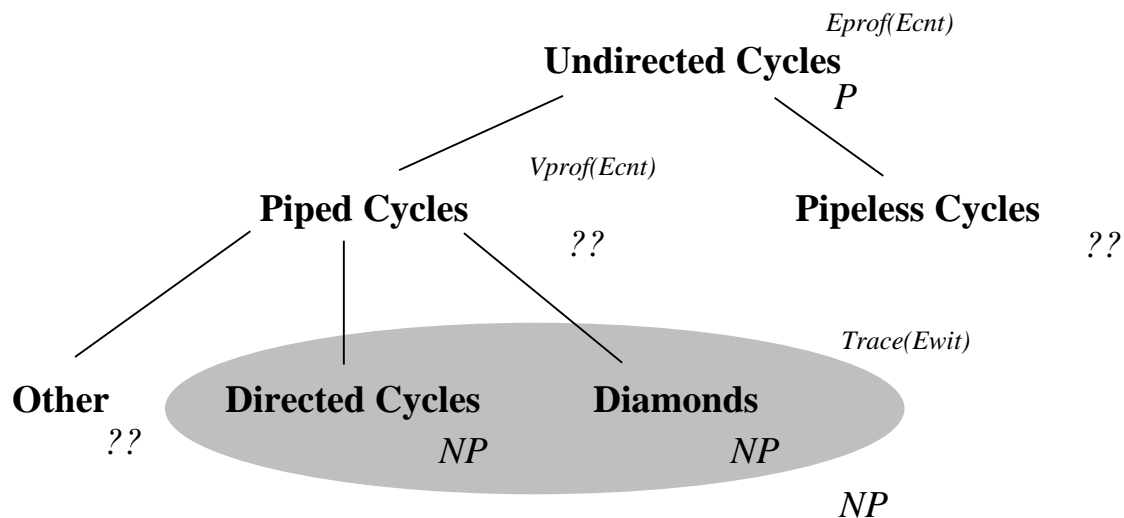
Figure 12. Hierarchy of cycles, the profiling or tracing problems they correspond to, and time complexity for breaking all cycles of a given type (*P* = polynomial; *NP* = NP-complete; ?? = unknown).

complete (Uniconnected Subgraph [9, 20]). Minimizing with respect to a weighting (that satisfies Kirchoff's flow law) does not make either of these problems easier. Furthermore, it is easy to show that optimally breaking both directed cycles and diamonds is no easier than either problem in isolation. Solving the tracing problem so that the cost of the instrumented edges is minimized is an NP-complete problem, as shown in an unpublished result by S. Pottle [24]. The reduction is similar to that used by Maheshwari but is complicated by the requirement that a weighting satisfies Kirchoff's flow law.

We believe that optimally solving *Vprof* (*Ecnt*) (minimizing the size or cost of *Ecnt*) is an NP-complete problem, but do not have a proof as of yet. We have shown that a related problem, finding a minimum size set of edges that breaks all pipes, is NP-complete. Breaking all pipes guarantees that all piped cycles will be broken, but not necessarily optimally (as it is possible to break all piped cycles without breaking all pipes).

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 executed basic blocks. The *tracing problem* is to record enough information about a program's execution to reproduce the entire execution. A straightforward way to solve this problem is to instrument each basic block so whenever it executes, it writes a unique token (called a *witness*) to a trace file. In this case, the trace file need only be read to regenerate the execution. A more efficient method is to write a witness only at basic blocks that are targets of predicates [17]. The following code regenerates the execution from a predicate trace file and the program's CFG *G*:

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

Assuming a standard representation for witnesses (*i.e.,* a byte, half-word, or word per witness), the tracing problem can be solved with significantly less time and storage overhead than the above solution by writing witnesses when edges are traversed (not when vertices are executed) and carefully choosing the witnessed edges. Section 4.1 formalizes the tracing problem for single-procedure programs. Section 4.2 considers complications introduced by multi-procedure programs.

4.1. Single-Procedure Tracing

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

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

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

$$witness\,(G,\ Ewit,\ p,\ q)\ =$$

$$\{\ w\ |\ p \rightarrow q\ \in\ Ewit\ (\text{and writes witness } w)\ \}$$

$$\cup\ \{\ w\ |\ x \rightarrow y\ \in\ Ewit\ (\text{and writes witness } w)\ \text{and } \exists\ \text{witness-free path } p \rightarrow q \rightarrow \cdots \rightarrow x\ \}$$

$$\cup\ \{\ EOF\ |\ \exists\ \text{witness-free path } p \rightarrow q \rightarrow \cdots \rightarrow EXIT\ \}$$

Figure 13 illustrates these definitions. We use $witness\,(p,\ q)$ as an abbreviation for $witness\,(G,\ Ewit,\ p,\ q)$.

Let us examine how the execution in Figure 13 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. Witness **u** is a member of $witness\,(C,P)$ but not $witness\,(C,EXIT)$, so vertex P follows C. At vertex P, witness **u** is still valid (since the edge $B \rightarrow A$ has not been traversed yet) and determines B as P's successor. Continuing in this manner, the original execution can be reconstructed.

If a witness w is a member of both $witness\,(G,\ Ewit,\ p,\ a)$ and $witness\,(G,\ Ewit,\ p,\ b)$, where $a \neq b$, then two *different* executions of G may generate the same trace, which makes regeneration based solely on control-flow and trace information impossible. For example, in Figure 13, if the edge $P \rightarrow A$ did not generate a witness, then $witness\,(P,A) = \{\ \textbf{u, v, EOF}\ \}$ and $witness\,(P,B) = \{\ \textbf{u, v}\ \}$. The executions $(P,\ A,\ C,\ P,\ B,\ C,\ EXIT)$ and $(P,\ B,\ C,\ EXIT)$ both generate the trace $(\textbf{v, EOF})$. This motivates our definition of the tracing problem:
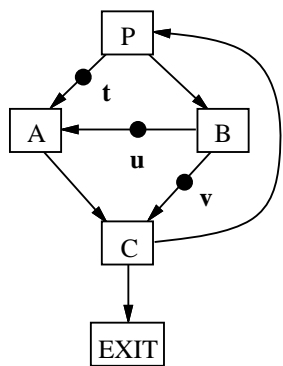
Execution: P A C P B A C P B C EXIT

Trace: t   u   v   EOF

$witness(P, A) = \{\ \textbf{t}\ \}$ $witness(B,A) = \{\ \textbf{u}\ \}$

$witness(P, B) = \{\ \textbf{u, v}\ \}$ $witness(B, C) = \{\ \textbf{v}\ \}$

$witness(C, P) = \{\ \textbf{t, u, v}\ \}$

$witness(C, EXIT) = \{\ \textbf{EOF}\ \}$

Figure 13. Example of a traced function. Vertices *P, B,* and *C* are predicates. The witnesses are shown by labeled dots on edges. For the execution shown, the trace generated is **(t, u, v, EOF)**. The witness **EOF** is always the last witness in a trace. The execution can be reconstructed from the trace using the witness sets to guide which branches to take.

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

It is straightforward to show that *Ewit* solves *Trace*(*Ewit*) for CFG *G* *iff* *E*−*Ewit* contains no diamonds or directed cycles. Optimally breaking diamonds and directed cycles is an NP-complete problem, as discussed in Section 3.4. Note that any solution to *Eprof*(*Ecnt*) or *Vprof*(*Ecnt*) is also a solution to *Trace*(*Ewit*), as breaking all undirected cycles or all simple piped cycles is guaranteed to break all directed cycles and diamonds. Edges not in the maximum spanning tree of the CFG comprise *Ewit* and solve *Trace*(*Ewit*) (but not necessarily optimally). However, for any CFG *G* in G^*, an optimal solution to *Eprof*(*Ecnt*) is also an optimal solution to *Trace*(*Ewit*) (because all directed cycles and diamonds are piped cycles and every cycle in a CFG from G^* is piped).

Given a CFG *G*, a set of edges *Ewit* that solves *Trace*(*Ewit*), and the trace produced by an execution *EX*, the algorithm in Figure 14 regenerates the execution *EX*.

4.2. Multi-Procedure Tracing

Unfortunately, tracing does not extend as easily to multiple procedures as profiling. There are several complications that we illustrate with the CFG in Figure 13. Suppose that basic block *B* contains a call to procedure *X* and execution proceeds from *P* to *B*, where procedure *X* is called. After *X* returns, suppose that *C* executes. This call creates problems for the regeneration process since the witnesses generated by procedure *X* and the procedures it invokes, possibly an enormous number of them, precede witness **v** in the trace file.

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

Figure 14. Algorithm for regenerating an execution from a trace.

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

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

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

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

The regeneration algorithm in Figure 14 need only be modified to maintain a stack of currently active procedures. When the algorithm encounters a call vertex, it pushes the current CFG name and *pc* value onto the stack and starts executing the callee. When the algorithm encounters an *EXIT* vertex, it pops the stack and resumes executing the caller.

An easy way to ensure that *Ewit* has the blocking property is to include each incoming edge to a call or *EXIT* vertex in *Ewit*. Figure 15 illustrates why this approach is suboptimal. The shaded vertices (*B*, *I*, and *H*) are call vertices. In the first subgraph, a blocking witness is placed on each incoming edge to a call vertex (black dots). In addition, a witness is needed on edge *B*→*D* (white dot). This placement is suboptimal because the witness on edge *H*→*I* is not needed, and because the witnesses on edges *B*→*D* and *G*→*I* (with cost = 3) can be replaced by witnesses on edges *B*→*D* and *B*→*E* (with cost = 2). In the second subgraph, blocking witnesses are placed as far from call vertices as possible, resulting in an optimal placement.

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

---

[3]In some tracing applications, data other than witnesses (such as addresses) are also written to the trace file. Vertices in the CFG that generate addresses can be blocked with witnesses so that no address is ever mistakenly read as a witness. It would also be feasible in this situation to break the trace file into two files, one for the witnesses and the other for the addresses, to avoid placing more blocking witnesses.
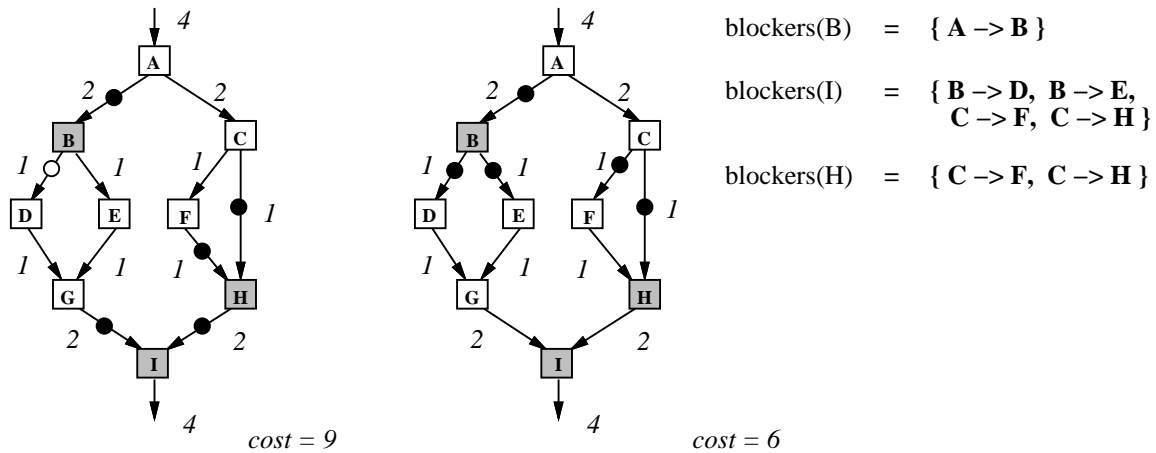
Figure 15. Two placements of blocking witnesses: a suboptimal placement and an optimal placement.

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

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

blockers(G, v) = { $p {\rightarrow} x_0$ | there is a path $p {\rightarrow} x_0 {\rightarrow} \cdots {\rightarrow} x_n$ where p is a predicate,

$v = x_n$, and for $0 \le i < n$, x_i is not a predicate }

First, for each vertex *v* that is a call or *EXIT* vertex, all edges in blockers(G, v) are added to *Ewit* (which is initially empty). To ensure that *Ewit* solves *Trace*(*Ewit*), we must add additional edges to *Ewit* so that $E - Ewit$ contains no diamonds or directed cycles. The maximum spanning tree algorithm can be modified to add these edges. No edge that is already in *Ewit* is allowed in the spanning tree.[4] Edges that are not in the spanning tree are added to *Ewit*, which guarantees that *Ewit* solves *Trace*(*Ewit*). Applying this algorithm to the control-flow fragment in Figure 16(a), the blocking phase adds the black dot edges to *Ewit*. The spanning tree phase adds the white dot edge to *Ewit*.

One might question whether it is better to reverse the above process and first compute an *Ewit* that solves *Trace*(*Ewit*), using the maximum spanning tree algorithm, and add blocking witnesses as needed afterwards. Figure 16(b) shows that this approach can yield undesirable results. The black dot edges are placed by the spanning tree phase and solve *Trace*(*Ewit*) but do not satisfy the blocking property. The white dot edge must be added to satisfy the blocking property and creates a suboptimal *Ewit*.

---

[4]The modified spanning tree algorithm may not actually be able to create a spanning tree of *G* because of the edges already in *Ewit*. In this case the algorithm simply identifies the maximal cost set of edges in $E - Ewit$ that contains no (undirected) cycle.
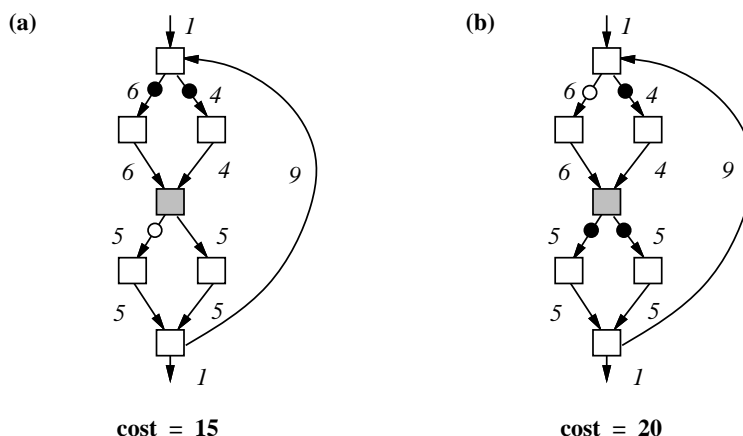
Figure 16. Ordering of blocking witness placement and spanning tree placement affects optimality.

## 5. A HEURISTIC WEIGHTING ALGORITHM

In order to profile or trace efficiently, instrumentation code should be placed in areas of low execution frequency. It may appear that to find areas of low execution frequency requires profiling. However, structural analysis of the CFG can often accurately predict that some portions are less frequently executed than others. This section presents a simple heuristic for weighting edges, based solely on control-flow information. As shown in Section 6, this simple heuristic is quite effective in reducing instrumentation overhead. The basic idea is to give edges that are more deeply nested in conditional control structures lower weight, as these areas will be less frequently executed. In general, every path through a loop requires instrumentation. However, within a loop containing conditionals, we would still like instrumentation to be as deeply nested as possible. For the CFG in Figure 17, the heuristic will generate the weighting shown in case (a). Any weighting of a CFG (*i.e.*, edge frequencies satisfying Kirchoff's flow law) that assigns each edge a non-zero weight will give edges that are more deeply nested lower weight. As discussed in Section 7, there are expensive matrix-oriented methods for generating weightings. Our heuristic has the advantage that it requires only a depth-first search and topological traversal of the CFG.

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

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

A vertex is a loop-entry if it is the target of one or more backedges. The natural loop of a loop-entry y, denoted by $nat-loop(y)$, is simply the union of all 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 to define the loop-exit edges of a loop with entry y:
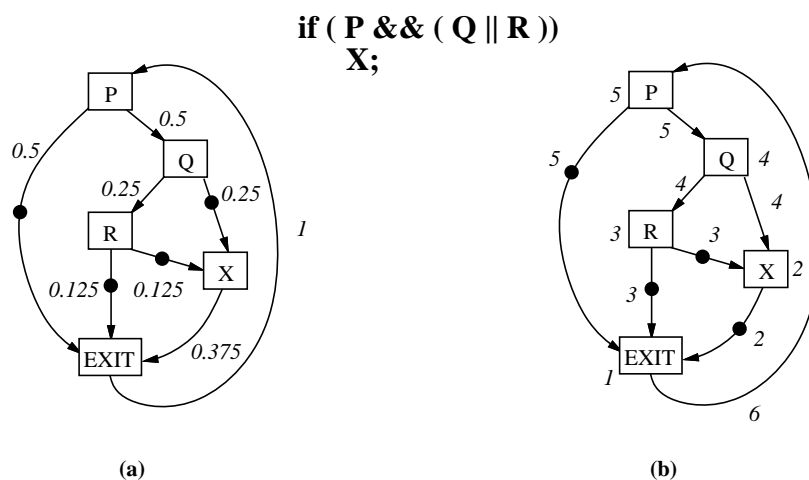
Figure 17. A program fragment, (a) its CFG with a weighting satisfying Kirchoff's flow, and an optimal edge counter placement (black dots). Case(b) shows a weighting derived using a post-order numbering of vertices (an edge's value is the post-order number of its source vertex), and the sub-optimal placement that results from finding a maximum spanning tree with respect to this weighting.

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

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

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

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

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

(3)   If v is a loop-entry vertex then let W be the weight of vertex v times *LOOP_MULTIPLIER*, otherwise let W be the weight of vertex v. If W_E is the sum of the weights of the outgoing edges of v that are loop-exit edges, then each outgoing edge of v that is not a loop-exit edge has weight $(W - W_E)/N$, where N is the number of outgoing edges of v that are not loop-exit edges.

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

6. PERFORMANCE RESULTS

This section describes several experiments that demonstrate that the algorithms presented above significantly reduce the cost of profiling and tracing real programs. Sections 6.1 and 6.2 discuss the performance of the profiling and tracing algorithms, respectively. Section 6.3 considers some optimizations that can further decrease the overhead of profiling and tracing. Section 6.4 examines the effectiveness of the heuristic weighting algorithm.

6.1. Profiling Performance

We implemented the profiling counter placement algorithm in *qpt* [18], which is a basic block profiler similar to MIPS's *pixie* [31]. *Qpt* instruments object code and can either insert counters in every basic block in a program (*redundant* mode) or along the subset of edges identified by the spanning tree algorithm (*optimal* mode).

We used the SPEC benchmark suite to test *qpt* [6]. This is a collection of 10 moderately large Fortran and C programs that is widely used to evaluate computer system performance. The programs were compiled at a high level of optimization (either -O2 or -O3, which does interprocedural register allocation). However, we did not use the MIPS utility *cord*, which reorganizes blocks to improve cache behavior, or interprocedural delay slot filling. Both optimizations confuse a program's structure and greatly complicate constructing a control-flow graph. Timings were run on a DECstation 5000/200 with local disks and 96MB of main memory. Times are elapsed times.

Table 1 describes the 10 benchmarks and shows the size of the object files and the time required to insert profiling code in redundant and optimal mode (keep in mind that *qpt* has not been tuned because its current speed is more than adequate for most executables encountered in practice). As can be seen, instrumenting for optimal profiling is slightly (22-38%) slower than instrumenting for redundant profiling. This is due to the extra work to find the loops in a CFG and to compute a weighting, to drive the maximum spanning tree algorithm. In practice, this extra instrumentation overhead is quickly regained from the reduction in profiling overhead.

Graph 1 shows the (normalized) execution time of the benchmarks without profiling, with *qpt* redundant profiling, with *pixie* profiling (which inserts a counter in each basic block), and with *qpt* optimal profiling.

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

Table 1. SPEC benchmarks. Size of input object files and times for instrumenting programs for Redundant and Optimal profiling. The first four programs are C programs. The remainder are FORTRAN programs.

Pixie rewrites the program to free 3 registers, which enables it to insert a code sequence that is almost half the size of the one used by *qpt* (6 instructions vs. 11 instructions). Of course, *pixie* may have to insert spill code in order to free registers.

As can be seen from Graph 1, Optimal profiling reduces the overhead of profiling dramatically over Redundant profiling, from 10-225% to 5-91%. These timings are affected by variations in instruction and data cache behavior caused by instrumentation. We measured profiling improvement in another way that factors out these variations. Graph 2 records the reduction in counter increments in going from Redundant to Optimal profiling (*i.e.*, the number of counter increments in Redundant mode / the number of increments in Optimal mode). The graph also records the reduction in number of instrumentation instructions executed (assuming 5 instructions for a counter increment and 1 instruction for an unconditional branch for the edge profiling code). In general, this reduction is less than the reduction in counter increments since edge profiling may require the insertion of unconditional jumps.

Fortunately, the greatest improvements occurred in programs in which profiling overhead was largest, since these programs had more conditional branches and more opportunities for optimization. For programs that frequently executed conditional branches, the improvements were large. For the four C programs (*cc1*, *espresso*, *xlisp*, and *eqntott*), the placement algorithm reduced the number of increments by a factor of 3-4 and the overhead by a factor of 2-3. For the Fortran programs, the improvements varied. In programs with large basic blocks that execute few conditional branches (where profiling was already inexpensive), improved counter placement did not have much of an effect on the number of increments or the
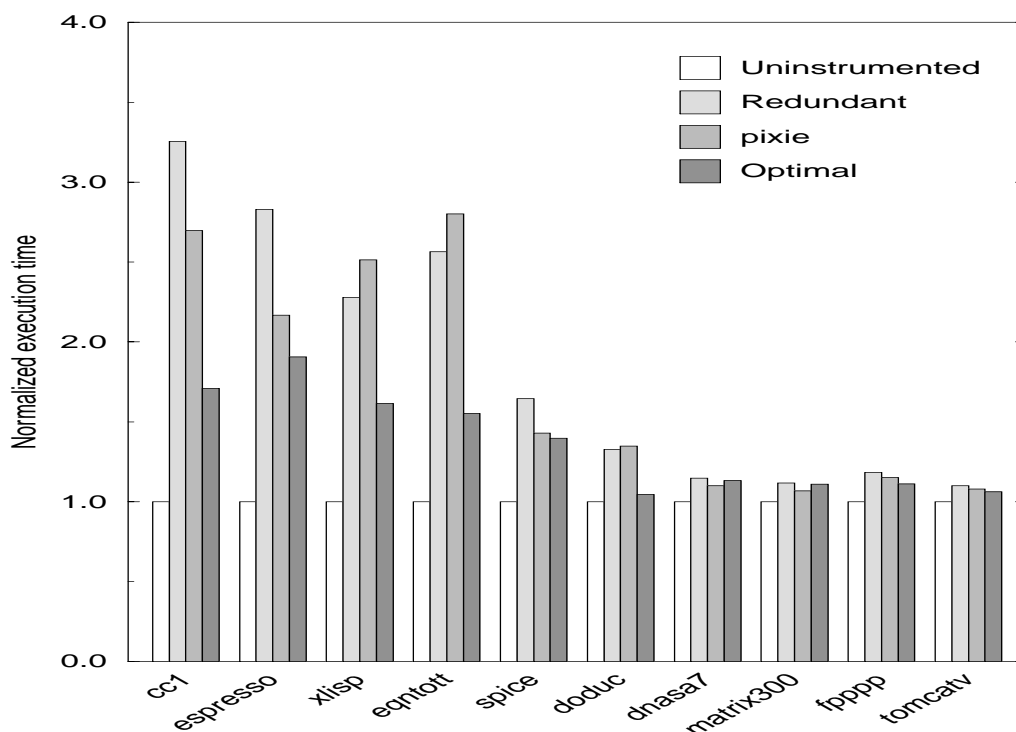

Graph 1. Normalized profiling execution times. For Redundant profiling, *qpt* inserts a counter in each basic block (vertex). For Optimal profiling, *qpt* inserts a counter along selected edges (*Eprof* (*Ecnt*)). *Pixie* is a MIPS utility that inserts a counter in each basic block.

cost of profiling. The FORTRAN program *doduc*, while it has a dynamic block size of 12.4 instructions, has "an abundance of short branches" [6] that accounts for its reduction in counter increments. The decrease in run time overhead for *doduc* was substantial (38% to 5%). The *fpppp* benchmark produced an interesting result. While it showed the largest reduction in counter increments, the overhead for measuring every basic block was already quite low at 18% and the average dynamic basic block size was 103.5 instructions. This implies that large basic blocks dominated its execution. Thus, even though many basic blocks of smaller size executed (yielding the dramatic reduction in counter increments), they contributed little to the running time of the program.

Graph 3 compares the reduction in dynamic instrumentation overhead for the *Eprof* (*Ecnt*) algorithm (optimal profiling), the *Vprof* (*Ecnt*) heuristic, and Knuth and Stevenson's *Vprof* (*Vcnt*) algorithm, as compared to redundant profiling (measure at every vertex). All algorithms used the same weighting to compute a counter placement. Given a counter placement for one of the algorithms, we used the profile information collected from a previous run to determine how many times each counter would have been incremented and how many extra jumps would have been needed (*Vprof* (*Vcnt*) does not require extra jumps since counting code is placed on vertices). By doing so, we avoided instrumenting and running the programs for
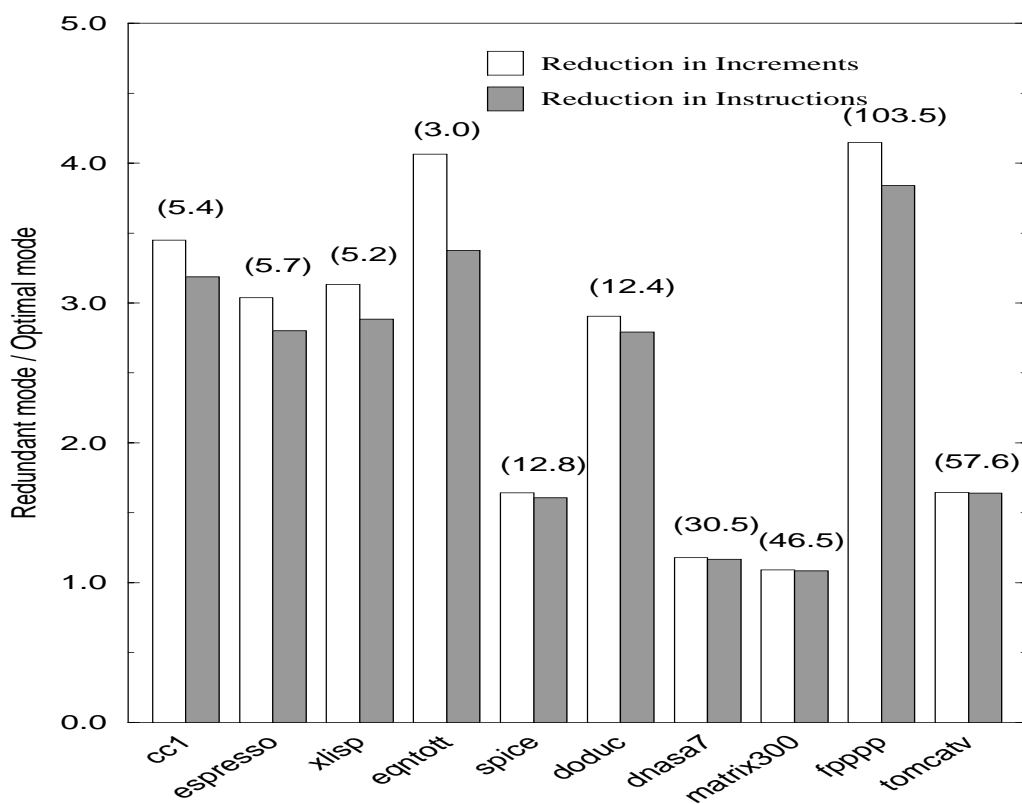
Graph 2. Reduction in counter increments and instrumentation instructions due to optimized counter placement, as guided by the heuristic weighting described in Section 5. Reduction in increments is (number of counter increments for Redundant profiling / number of counter increments for Optimal profiling). Reduction in instrumentation is (5 * number of basic blocks) / (5 * increments + number of extra jumps). The average dynamic basic block size (in instructions) for each program is shown in parenthesis.
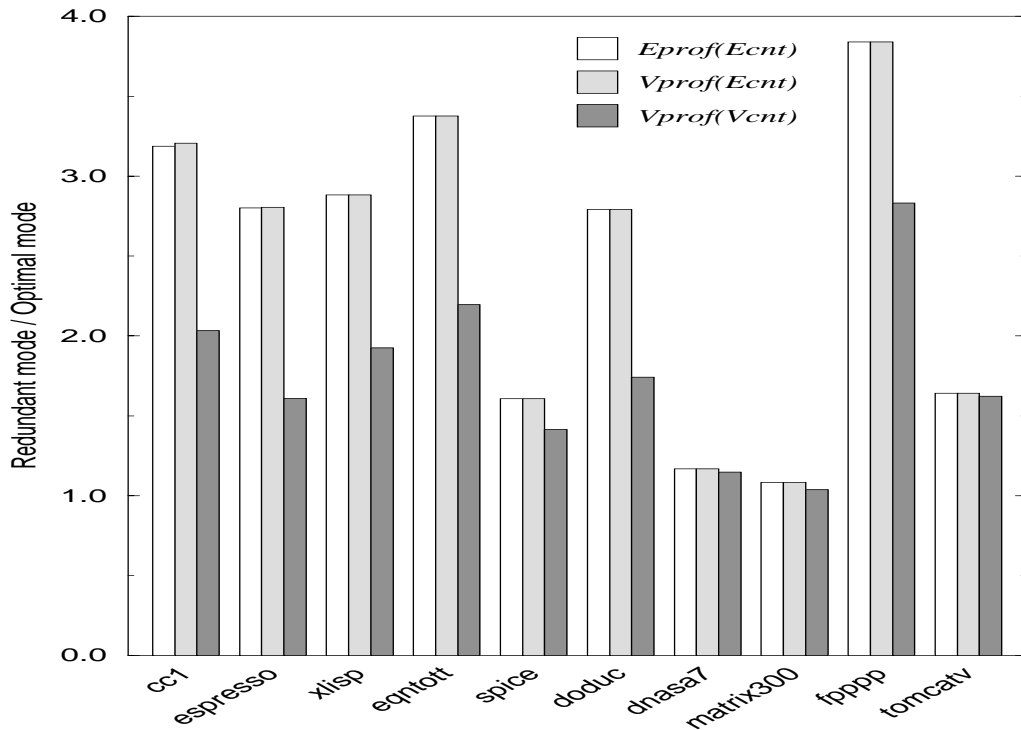
Graph 3. Comparison of instrumentation reduction of *Eprof* (*Ecnt*), *Vprof* (*Ecnt*), and *Vprof* (*Vcnt*). The larger a plot, the better (*i.e.*, the greater the reduction of instrumentation code). Instrumentation reduction = (5 * basic blocks) / (5 * increments + number of extra jumps).

every algorithm, while still collecting accurate results. For all the benchmarks, *Eprof* (*Ecnt*) is superior to *Vprof* (*Vcnt*), producing a greater reduction in instrumentation instructions, as predicted. The heuristic for *Vprof* (*Ecnt*) yields almost no improvement over *Eprof* (*Ecnt*), as there are very few cases when a counter can be eliminated.

Table 2 provides statistics on the number of edges in each program ("Total Edges"), the number of edges that had counting code placed on them using the spanning tree algorithm ("Profiled Edges"), and the number of profiled edges that did not require the insertion of an unconditional jump ("No-Jump Edges"). We make two observations. First, notice that the percentage of all edges that are profiled is in the narrow range of 39-46%. This is consistent with the facts that most CFGs have almost (but not quite) twice as many edges as vertices and that the number of edge counters required for edge profiling is $|E| - (|V| - 1)$. Second, less than half of all profiled edges require the insertion of an unconditional jump.

6.2. Tracing Performance

The witness placement algorithm was implemented in the AE program-tracing system [17], which has since been incorporated as part of the *qpt* tool. AE originally recorded the outcome of each conditional branch and used this record to regenerate a full control-flow trace. One complication is that AE traces both the instruction and data references so a trace file contains information to reconstruct data addresses as well as the witnesses. Combining this information in one file requires additional blocking witnesses, as described in Section 4.2.

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

Table 2. Static statistics on control-flow edges. "Total Edges" shows the total number of control-flow graph edges in each program. "Profiled Edges" shows the number of edges that had counters placed on them using the spanning tree algorithm. "No-Jump Edges" shows the number of profiled edges that do not require the insertion of an unconditional jump.

Table 3 shows the reduction in total file size ("File"), witness trace size ("Trace"), and execution time that result from switching the original algorithm of recording each conditional ("Old") to the witness placement described in Section 4 ("New"). As with the profiling results, the programs with regular control-flow, *sgefa* and *pdp*, do not gain much from the tracing algorithm. For the programs with more complex control-flow, *compress* and *polyd*, the tracing algorithm reduced the size of the trace file by factors of 3 and 2.7 times, respectively.

In the discussion of tracing we assumed that a standard representation was used for witnesses (per CFG). In modern architectures it is convenient for this representation to be a multiple of a byte. Thus, it is often the case that we record more bits per witness than necessary. We explored another method for tracing, called *bit tracing*, which seeks to reduce the size of the trace. With bit tracing, each outgoing edge of a predicate vertex generates a witness and witness values are reused. For predicates with two successors, only one bit of information is required to distinguish its witness sets. In general, a predicate with N successors requires $\log_2 N$ bits. Figure 18 illustrates the tradeoff between the spanning tree approach and bit tracing. In case (a), witnesses are placed according to the spanning tree approach. No pair of distinct witnesses from the set **{ a, b, c, d }** can be assigned the same value, so two bits per witness are required. In case (b), only one bit per witness is required. Any iteration of the loop in this CFG will generate three bits of trace. However, in case (a) the amount of trace generated per iteration can either be two or four bits. In

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. Old refers to the original version of AE, which recorded the outcome of every conditional branch. New refers to the improved version of AE, which uses the witness placement algorithm of Section 4. 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.

this example, neither witness placement is a clear winner.

If compared to the spanning tree approach that naively uses a byte (or more, if needed) of storage per witness, bit tracing is clearly superior. Although more instrumentation code is executed, less trace is generated, which reduces I/O overhead. This method decreases the size of the trace 3-7 times over the spanning tree approach. However, as shown in Figure 18, by using only as many bits as necessary, the spanning tree approach can be improved upon. In this example, two bits per witness are needed. In general, if there are N witnesses for a CFG then at most $\log_2 N$ bits per witness are needed. However, there are situations where witness values can be reused, possibly decreasing the number of bits needed. This is complicated by the fact that different placements of witnesses may give rise to different opportunities for the reuse of values. Further investigation in optimizing the spanning tree approach is clearly needed.

Bit tracing avoids the multi-procedure tracing problem discussed in Section 4.2 as there is no witness-free directed path from a predicate to a call vertex. If an address trace also is generated from the program, bit tracing requires that two separate files be maintained (for efficiency), one for the instruction trace and one for the address trace. The cost of bit tracing is the additional implementation complexity required to manage witnesses at the bit level.

6.3. Optimizations

Several optimizations can further decrease the overhead of profiling and tracing. The first optimization, *register scavenging*, is specific to instrumenting object code. For RISC machines, counter increment code requires two registers, one to hold the counter's address (because addressing on RISC machines is done by indirection off of a register) and one to hold the counter's value. If both registers need to be saved and restored (to preserve their values), the instrumentation code jumps from 5 to 11 instructions. Register scavenging notes the unused caller-saved registers in a procedure. These registers can be used by
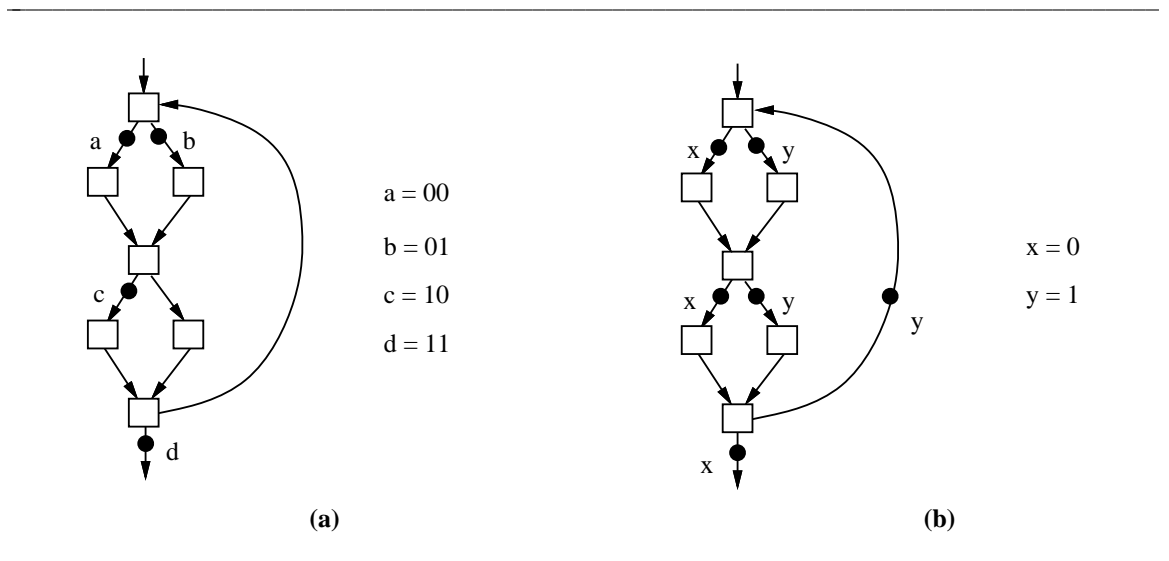

Figure 18. Tradeoff between (a) placing witnesses according to the spanning tree approach and (b) placing witnesses on every outgoing edge of a predicate vertex.

instrumentation code without preserving their values, since the procedure's callers expect these registers to be modified.[5] For many of the benchmarks, specifically the FORTRAN programs with large basic blocks and few unused registers, register scavenging had little effect on execution overhead. For other benchmarks, the results varied from small reductions of a few percent to larger reductions in the range of 6-21%.

The second optimization can substantially reduce profiling overhead by removing counters from loops. If the number of iterations of a loop can be determined before the loop executes or from an induction variable whose value is recorded before and after the loop, then a counter can be eliminated from the loop body (allowing one counter-free path through the loop). Both Sarkar and Goldberg have successfully implemented this approach in profiling tools [10, 29]. For example, Goldberg reports that for *eqntott* the reduction in increments (redundant/optimal profiling) increased from 4.3 to 7.7 after adding induction variable analysis. Some scientific codes benefitted greatly from this analysis (a 33-fold decrease in instrumentation code executed for *matrix300*). However, for pointer-chasing programs such as *xlisp* the benefits of this analysis were quite small, as few induction variables are present in such programs.

As mentioned before, placing instrumentation code on edges may require the insertion of jumps in order to avoid executing other instrumentation code. For example, in the control-flow fragment of Figure 19(a) there are two instrumented incoming edges to a vertex. Because we use the general rule that the instrumentation code associated with an edge is placed just before the code associated with the vertex that is the target of the edge, this fragment will require at least one unconditional jump (in order to jump over the instrumentation code associated with the other edge). However, the grey vertex has only one incoming edge and only one outgoing edge, so the instrumentation point can be moved from its outgoing edge to its incoming edge, resulting in the placement in case (b). This placement may require no extra jumps (unless the grey vertex's outgoing edge is a fall-through). Jump optimization searches for vertices with one incoming and one outgoing edge with instrumentation code on the outgoing edge. The instrumentation code is simply moved to the incoming edge. This simple optimization may reduce (and will never increase) the number of extra jumps needed. In the case of *xlisp*, this optimization reduced execution overhead by 10 percent.
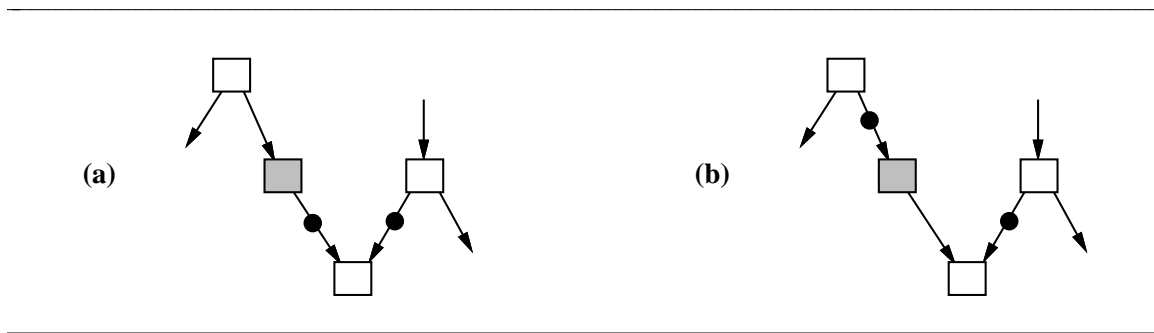
Figure 19. (a) Placement requiring insertion of jump. (b) No jumps required.


[5]We discuss the problems of register scavenging and instrumenting object files in greater detail elsewhere [19].

6.4. Effectiveness of the Heuristic Weighting Algorithm

The effectiveness of the heuristic weighting algorithm was measured in two ways, as presented in Graph 4. First, we measured the reduction in counter increments (number of increments in Redundant mode / number of increments in Optimal mode) using an exact edge weighting from a previous run of the same program with identical input. This number, "Exact-Max", represents the best one could hope to do without semantics-based optimizations (such as induction variable analysis). Second, for both the heuristic and exact weightings, we also computed what the reduction in increments would be if a minimum spanning tree were used to place counters. While the maximum spanning tree places counters in less frequently executed areas of the CFG, a minimum spanning tree places counters in more frequently executed areas. Thus, "Exact-Min" is the worst possible reduction for the spanning tree algorithm. As the difference between "Exact-Min" and "Exact-Max" shows, there is great variation in the reduction in counter increments, depending on which spanning tree is chosen. The heuristic is clearly successful at predicting areas of low execution frequency, as shown by "Heuristic-Max". Also, note that "Heuristic-Max" always produced a greater reduction than "Heuristic-Min". The difference in reduction between the heuristic and exact weightings was usually small (ranging from 1% to 34%). Not surprisingly, the heuristic was quite accurate for the FORTRAN programs with few conditional branchs.

Graph 5 shows the normalized times for the benchmarks run under Optimal profiling for the heuristic weighting (corresponds to "Heuristic-Max" in Graph 4) and exact weighting (corresponds to "Exact-Max" in Graph 4). In one case (*fpppp*), the run time with the exact weighting is greater than the run time with the
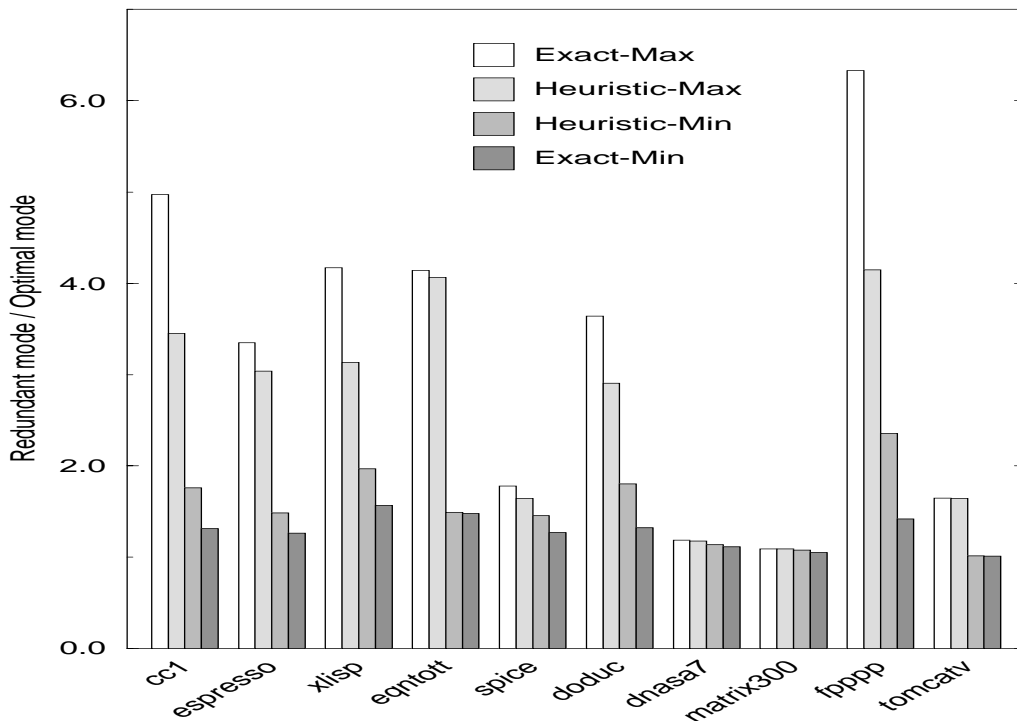
Graph 4. Heuristic weighting vs. exact weighting: reduction in increments (Redundant/Optimal). Reductions in increments are computed for counter placements from both maximum and minimum spanning trees.
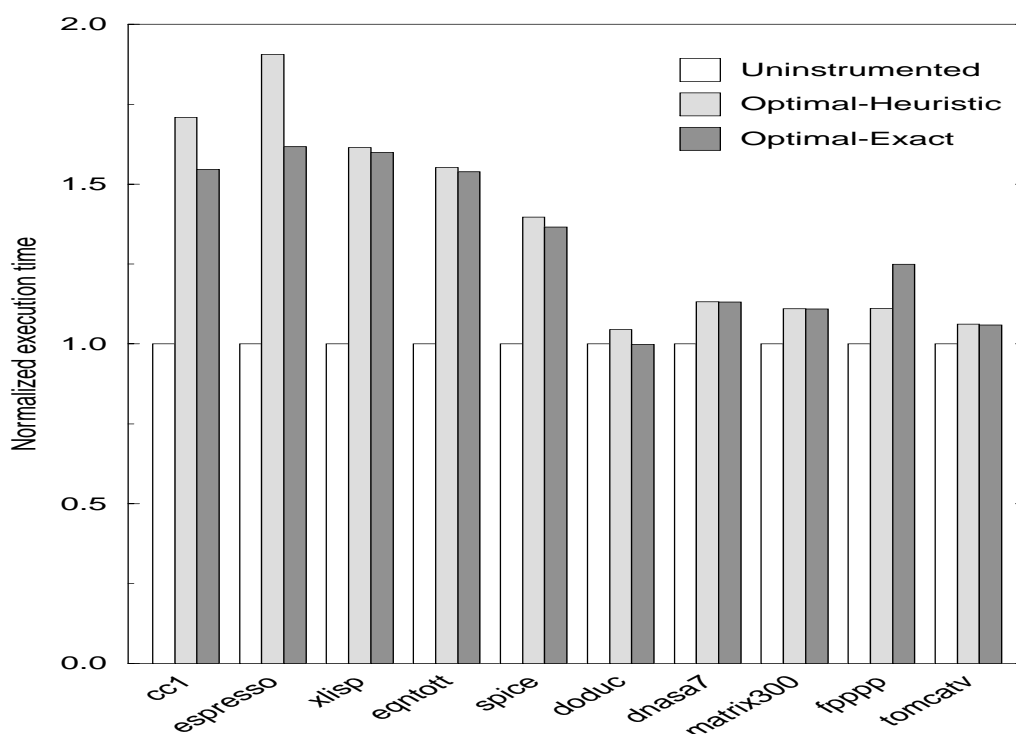
Graph 5. Normalized profiling times for heuristic and exact weightings.

heuristic weighting. Such an aberration is most likely due to different instruction and data cache behavior of the instrumented program under the different counter placements, and requires further investigation.

The heuristic weighting algorithm assumes each branch of a predicate is equally likely to be chosen. For most programs, varying this probability does not have a great effect on instrumentation overhead. However, weighting schemes that attempt to pick likely branch directions independently may have greater success. For example, favoring edges leading to blocks containing loops (which have a high dynamic cost) reduces instrumentation overhead for a few of the benchmarks.

7. RELATED WORK

7.1.1. Edge Profiling
The spanning tree solution to *Eprof* (*Ecnt*) has been known for a long time. In the area of network programming, the problem is known as the specialization of the simplex method to the network program [13]. Knuth describes how to use the spanning tree for profiling in [15]. Other authors that have written about the application of the spanning tree to profiling include Goldberg [10], Samples [28], and Probert [25]. As far as we know, Goldberg and Samples are the only other researchers that have implemented the spanning tree approach and performed significant experimentation with real programs. Their work occurred concurrently with ours.

Goldberg implemented edge profiling by instrumenting executable files [10]. His profiler was built as part of a system to analyze the memory performance of programs [11]. Goldberg optimized his instrumentation in two ways that we do not consider. First, his tool selected the two statically least-used registers in the executable and eliminated all uses by inserting loads and stores around existing uses of these registers.

This allows every counting code sequence to use these registers without saving and restoring them. A similar approach is used by MIPS's *pixie* profiling tool [31]. As a result, the number of instructions needed to increment a counter in memory can be cut roughly in half. Our tool only looks for free registers to scavenge and often must save and restore registers in the counter increment code sequence. Second, Goldberg identifies simple loop induction variables. This allows a counter to be eliminated from a loop (because the number of iterations can be inferred from the beginning and ending values of the induction variable), lowering instrumentation overhead drastically for scientific codes. Our tool does not perform this optimization.

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

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

Sarkar describes how to choose profiling points using control dependence and has implemented a profiling tool for the PTRAN system [29]. His algorithm finds a minimum sized solution to *Eprof* (*Ecnt*) based on a variety of rules about control dependence, as opposed to the spanning tree approach. There are several other major differences between his work and our work: (1) His algorithm only works for a subclass of reducible CFGs; (2) His algorithm does not use a weighting to place counters at points of lower execution frequency. As a result, the algorithm may produce suboptimal solutions; (3) When the bounds of a **DO** loop are known before execution of the loop, his algorithm eliminates the loop iteration counter, as done by Goldberg.

7.1.2. Vertex Profiling

Knuth and Stevenson exactly characterize when a set of vertices *Vcnt* solves *Vprof* (*Vcnt*) and show how to efficiently compute a minimum size *Vcnt* that solves *Vprof* (*Vcnt*) [16]. The authors note that their algorithm can be modified to compute a minimum cost solution to *Vprof* (*Vcnt*) given a set of measured or estimated vertex frequencies. Our work shows that it is less costly to measure vertex frequency by instrumenting edges rather than vertices.

7.1.3. Tracing

Ramamoorthy, Kim, and Chen consider how to instrument a single-procedure program with a minimal number of monitors, so the traversal of any directed path through the program may be ascertained after an execution [26]. This is equivalent to the tracing problem for single-procedure programs discussed here. The authors do not give an algorithm for reconstructing an execution from a trace or consider how to trace multi-procedure programs. Further, they are interested in finding a minimal *size* solution to the tracing problem, an NP-complete problem [20]. However, a minimum size solution does not necessarily yield a minimum cost solution.

7.1.4. Minimizing instrumentation overhead

A CFG has many spanning trees, each of which induces a counter placement with an associated run-time overhead cost. Section 5 presented our heuristic for estimating edge frequency in order to drive the maximum spanning tree algorithm. This section compares our heuristic to other methods for minimizing instrumentation overhead (which may include methods for estimating frequency). We use the CFG in Figure 17(a) as a basis for comparing the various heuristics discussed below. The weighting of this CFG satisfies the flow law and the edges with black dots are an optimal edge counter placement for profiling (with respect to this weighting). The other edges form a maximum spanning tree. As mentioned before, our heuristic generates the weighting in case (a).

Forman discusses the problem of minimizing counter overhead with the spanning tree approach from a graph theoretic perspective [8]. He defines a partial order on the spanning trees of a CFG such that for any weighting, if a spanning tree T is not a least element in the partial order then there is some spanning tree lower in the order that induces a counter placement with lower cost than the one induced by T. Of course, there may be more than one least element in the partial order. The spanning tree in Figure 17(a) is a least element. Forman proposes a structural method for computing a least element, but it works only for structured CFGs. Our heuristic works for any CFG. He also proposes a more general solution that generates a weighting, given branch probabilities for the predicate vertices in any CFG. A maximum spanning tree found under a weighting is a least element in Forman's partial order. To generate the weighting requires matrix operations on what are essentially adjacency matrix representations of the CFG. As such, this general approach would be much slower than our heuristic, which operates directly on the control-flow graph structure. Our heuristic generates edge frequencies satisfying the flow law and can easily be adapted to take branch probabilities into account.

Goldberg developed a heuristic for his profiling tool that uses a post-order numbering of the vertices in the CFG (as determined by a depth-first search from the *root* vertex) to assign edge weights [10]. He defines an edge's weight to be the post-order number of its source vertex. However, if an edge is a loop backedge then it is given a weight larger than the number of vertices in the graph. The rationale for this heuristic is that "...a node always executes at least as many times as any of its descendants [successors]; hence, it seems best to place counters on nodes as far from the root as possible." This heuristic clearly does not produce a weighting satisfying the flow law, as Figure 17(b) shows. Because the distance of an edge from the root vertex does not always correspond to that edge's level of nesting, Goldberg's heuristic will not always lead to the best counter placements. In the example of Figure 17(b), the maximum spanning tree for the given weighting (determined by his heuristic) induces a sub-optimal counter placement.

Wall experimented with a number of heuristics for estimating basic block and procedure profiles solely from program text, reporting poor results [33]. Wall's heuristics use information about loop nesting and call graph structure to predict basic block and procedure profiles, but do not take into account conditional control-flow (*i.e.*, predicting that code that is more deeply nested in conditionals is executed less frequently), as our heuristic does. It is this aspect of our heuristic that is key to reducing instrumentation overhead (this is also the main idea behind Forman's partial order). With Wall's heuristic, every basic block nested in the same number of loops gets equal weight. In the example graph of Figure 17, each block would get equal weight, which is clearly not useful for the purposes of minimizing instrumentation cost.

Other authors have presented heuristics that are similar to ours, usually for the purpose of aiding code optimization. For example, Fisher, Ellis, Ruttenberg, and Nicolau use loop nesting level and programmer-

supplied hints to estimate block execution frequency for trace scheduling [7]. However, few of these heuristics have the goal of producing edge frequencies satisfying the flow law.

None of the heuristics mentioned above nor our heuristic attempts to predict branch directions. If branches can be accurately predicted, then instrumentation code can be placed on the less frequently executed branch when a choice is possible. More recent work on branch prediction by Ball and Larus could be used in this application [3].

8. CONCLUSIONS

This paper studied algorithms for efficiently profiling and tracing programs. These algorithms optimize placement of instrumentation code with respect to a weighting of the control-flow graph. Empirical results on real programs show that these algorithms are successful in reducing instrumentation overhead. Placing instrumentation code along edges in the control-flow graph is essential to reduce both profiling and tracing overhead. However, several open questions remain: (1) Is there an efficient algorithm to optimally solve the vertex frequency problem with a set of edge counters or is the problem intractable? (2) Are there better weighting schemes that can more accurately guide the placement of instrumentation code?

ACKNOWLEDGEMENTS

REFERENCES

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

2. T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages,* (Albuquerque, NM, January 19-22, 1992), pp. 59-70 ACM, (1992).

3. T. Ball and J. R. Larus, "Branch prediction for free," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (published in SIGPLAN Notices)* **28**(6) pp. 300-13 ACM, (June 1993).

4. J. D. Choi, B. P. Miller, and R. H. B. Netzer, "Techniques for debugging parallel programs with flowback analysis," *ACM Transactions on Programming Languages and Systems* **13**(4) pp. 491-530 (October 1991).

5. 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).

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

7.   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).

8.   I. R. Forman, "On the time overhead of counters and traversal markers," *Proceedings of the 5th International Conference on Software Engineering*, pp. 164-169 (March 1981).

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

10.   A. Goldberg, "Reducing overhead in counter-based execution profiling," Technical Report CSL-TR-91-495, Stanford University, Stanford, CA (October, 1991).

11.   A. J. Goldberg and J. L. Hennessy, "Mtool: An integrated system for performance debugging shared memory multiprocessor applications," *IEEE Transactions on Parallel and Distributed Systems* **4**(1) pp. 28-40 (January 1993).

12.   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).

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

14.   B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall Software Series, Englewood Cliff, NJ (1978. Second edition, 1988).

15.   D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* Addison-Wesley, Reading, MA (1968. Second Edition: 1973).

16.   D. E. Knuth and F. R. Stevenson, "Optimal measurement points for program frequency counts," *BIT* **13** pp. 313-322 (1973).

17.   J. R. Larus, "Abstract execution: A technique for efficiently tracing programs," *Software—Practice and Experience* **20**(12) pp. 1241-1258 (December, 1990).

18.   J. R. Larus, "Efficient program tracing," *IEEE Computer* **26**(5) pp. 52-61 (May 1993).

19.   J. R. Larus and T. Ball, "Rewriting executable files to measure program behavior," *to appear in Software—Practice and Experience*, ().

20.   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).

21.   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).

22.   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).

23. 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).

24. S. Pottle, *private communication*. October 1991.

25. 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).

26. 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).

27. M. V. S. Ramanath and M. Solomon, "Optimal Code for Control Structures," pp. 82-94 in *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages,* (Albuquerque, NMP, ACM, New York (1982).

28. A. D. Samples, "Profile-driven compilation," Ph. D. Thesis (Report No. UCB/CSD 91/627), University of California at Berkeley (April 1991).

29. 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).

30. A. J. Smith, "Cache memories," *ACM Computing Surveys* **14**(3) pp. 473-530 (1982).

31. MIPS Computer Systems, Inc., *UMIPS-V Reference Manual (pixie and pixstats),* MIPS Computer Systems, Sunnyvale, CA (1990).

32. R. E. Tarjan, *Data Structures and Network Algorithms,* Society for industrial and applied mathematics, Philadelphia, PA (1983).

33. D. W. Wall, "Predicting program behavior using real or estimated profiles," *Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation,* (Toronto, June 26-28, 1991)*, ACM SIGPLAN Notices* **26**(6) pp. 59-70 (June, 1991).