Efficiently Counting Program Events with Support for On-Line Queries

THOMAS BALL
University of Wisconsin, Madison

The ability to count events in a program's execution is required by many program analysis applications. We present an instrumentation method for efficiently counting events in a program's execution, with support for on-line queries of the event count. Event counting differs from basic block profiling in that an aggregate count of events is kept rather than a set of counters. Due to this difference, solutions to basic block profiling are not well suited to event counting. Our algorithm finds a subset of points in a program to instrument, while guaranteeing that accurate event counts can be obtained efficiently at every point in the execution.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement Techniques; D.2.2 [**Software Engineering**]: Tools and Techniques

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Control-flow graph, counting, instrumentation

1. INTRODUCTION

The ability to count events in a program's execution is required by many program analysis applications. Instruction counts are used to determine how much time is spent in a procedure [Graham et al. 1983]; debuggers or execution-driven simulators use countdown timers to return control from an executing program to the debugger or simulator after a certain number of events [Mellor-Crummey and LeBlanc 1989; Reinhardt 1993]; counts of synchronization events, I/O events, and system calls are used to measure the performance of parallel programs [Hollingsworth and Miller 1993]. Many of these applications require the capability to query the event count on-line, while the program executes, rather than off-line, after the program has terminated.

We investigate how to count events in a program execution efficiently, with support for on-line queries of the event count. An algorithm for efficiently

This work was supported in part by the National Science Foundation under grant CCR-8958530, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from Xerox and 3M.
Author's address: AT & T Bell Laboratories, Software Production Research Department, Room 1G-359, P.O. Box 3013, 1000 E. Warrenville Rd. Naperville, IL 60566-7013; email: tball@research.att.com

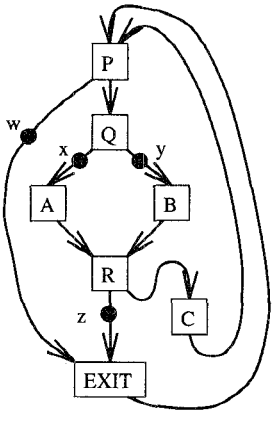
counting and querying program events is given that uses program instrumentation. Rather than instrument every program component, the algorithm instruments at select points, while guaranteeing that an accurate event count can be obtained efficiently at every point in the execution.

Event counting has a simple formalization. Each program component C (either a basic block or a control-flow edge) has a constant number of events, denoted by $Events(C)$. The goal of event counting is to track the total number of events in executed components. Because many applications (such as debuggers and simulators) require completely accurate event counts, we use program instrumentation to obtain the counts. There are two basic instrumentation methods for event counting:

(1) Instrument the program to record the number of times each component executes (i.e., maintain a basic block and/or edge profile). The event count is computed by summing the component counters, weighted appropriately (e.g., if component C executes i times then add $i * Events(C)$ to the event count). Because of the potentially large number of counters that must be summed for each query, this method is suitable only if there are few queries of the event count during the execution of the program, or the query is made off-line.
(2) Instrument the program to record the number of events directly in an event counter. A straightforward approach is to add code to each component C to increment the event counter by $Events(C)$. This approach can incur high overhead (in the range of 200–300%) for programs with small basic blocks [Ball and Larus 1994]. Furthermore, if instrumentation code is dynamically added to and deleted from programs by patching a basic block with a jump to a code stub rather than by rewriting the original code, the overhead can increase substantially [Kessler 1990].

We present an event-counting scheme that is similar to (2); however, it involves instrumenting control-flow edges (rather than basic blocks) in a procedure's control-flow graph. Previous work has shown that instrumentation of control-flow edges can be used to profile programs with low execution time overhead [Samples 1991; Goldberg 1991; Ball and Larus 1994]. Instrumentation of control-flow edges rather than basic blocks allows greater opportunity to place the code in areas of lower execution frequency.

In efficient profiling, each instrumented edge has its own counter, which is incremented whenever the edge executes. After program execution, a propagation phase determines the count for each uninstrumented edge from the instrumented edges' counts. In efficient event counting, each instrumented edge increments the same counter, but by different increments. Before program execution, event counts are propagated from uninstrumented to instrumented edges to determine the event increment for each instrumented edge. The challenge is to determine a necessary and sufficient set of edges to instrument, and to compute the increment associated with each instrumented edge.



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

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

Fig. 1. An example of efficient event counting.

Figure 1 gives an example of efficient event counting. In this example, the number of events in a vertex (basic block) is denoted by the vertex's label. The example control-flow graph has four instrumented edges (w, x, y, z), marked with black dots. For each instrumented edge e, there is a constant increment $Increment(e)$. This amount is added to the event counter each time edge e executes. It is not hard to verify that for any execution path from P to $EXIT$, the event count (as computed by the instrumented edges in the path) is correct. For example, the count for the path $P \rightarrow Q \rightarrow A \rightarrow R \rightarrow EXIT$ is $Increment(x) + Increment(z) = (+P + Q + A + R + C) + (-C + EXIT) = (+P + Q + A + R + EXIT)$.

An effect of efficiently counting events is that the event counter may not be accurate at all points. In Figure 1, if the path $P \rightarrow Q$ has executed, then the event counter has not been incremented at all. On the other hand, if the path $P \rightarrow Q \rightarrow A$ has executed, then the event count overestimates the true count at A since the counts for R and C have been incorporated. We solve this problem by computing for every vertex v a query increment $QueryInc(v)$ that, when added to the event counter, produces the correct count. Figure 1 shows the query increment for each vertex in the control-flow graph.

The uninstrumented edges in the control-flow graph of Figure 1 form an (undirected) spanning tree of the graph. We show that for any spanning tree of the control-flow graph, instrumentation of nontree edges is sufficient for event counting. A simple depth-first search algorithm computes the increment value for each nontree edge. There are also cases in which cycles of uninstrumented edges are allowed, as we will explore.

We first present some background material on control-flow graphs and spanning-tree theory. We then show how to count intraprocedural and interprocedural events. Last, we discuss related work, and conclude.

2. BACKGROUND

A *path* in a directed graph is a sequence of n vertices and $n - 1$ edges of the form $(v_1, e_1, v_2, \ldots e_{n-1}, v_n)$, where for each edge e_i, either $e_i = v_i \rightarrow v_{i+1}$ or

$e_i = v_{i+1} \rightarrow v_i$. A *cycle* is a path such that $v_1 = v_n$. A path or cycle is *directed* if for every edge e_i, $e_i = v_i \rightarrow v_{i+1}$. A *simple cycle* is a cycle in which every vertex occurs exactly once, except for vertex v_1 which occurs exactly twice ($v_1 = v_n$). Given an edge e, $src(e)$ and $tgt(e)$ denote the vertices that are the source and target, respectively, of e. We use the terms *path* and *cycle* to denote undirected paths and cycles. When edge direction matters, we state explicitly that a path or cycle is directed.

A control-flow graph is a rooted directed graph $G = (V, E)$ with a special vertex *EXIT* (distinct from the root vertex) that corresponds to a procedure in the following way: each vertex in V represents a basic block of instructions, and each edge in E represents the transfer of control from one basic block to another. The root vertex represents the entry point of the procedure and *EXIT* the return point. Every vertex is on a directed path from the root vertex to *EXIT*. For technical reasons, there is an edge $EXIT \rightarrow root$, which makes the graph strongly connected. The *EXIT* vertex has no successors other than the root vertex.

An execution of a procedure induces a directed path from the root vertex to the *EXIT* vertex. The addition of edge $EXIT \rightarrow root$ to the end of the directed path turns it into a directed cycle.

Although the control-flow graph is directed, *the spanning trees that we consider are undirected.* A spanning tree of a graph $G = (V, E)$ is a subgraph that is a tree and contains all the vertices of G. If T is the set of spanning-tree edges, then any edge in $E - T$ is called a *chord* of the spanning tree. The addition of any chord e to the spanning tree creates exactly one simple cycle. The cycle is called the *fundamental cycle* of e and is denoted by $C(e)$.

We will use Theorem 2.1 [Knuth 1968] which shows that the number of times an edge appears in a directed cycle in a control-flow graph is uniquely determined by the number of times each chord (of a spanning tree of the graph) appears in the cycle. As discussed in Section 5, this theorem has applications to the related area of profiling. Let $\#(P, f)$ denote the number of times edge f appears in path P. Given a cycle D containing edges e and f, let $Dir(D, e, f)$ equal 1 if edges e and f are in the same direction in D, and -1 if edges e and f are in opposite directions in D.

THEOREM 2.1 (KNUTH). *Let G be a control-flow graph; let T be a spanning tree of G and let D be a directed (not necessarily simple) cycle in G. For all edges e:*

$$\#(D, e) = \sum_{\substack{\forall f \in E - T \\ s.t.\ e \in C(f)}} \#(D, f) Dir(C(f), e, f).$$

3. INTRAPROCEDURAL-EVENT COUNTING

This section focuses on counting edge events within one procedure. Vertex events can be modeled as edge events by replacing a vertex v by an edge $v_{in} \rightarrow v_{out}$, where the predecessors of v become predecessors of v_{in}, and the successors of v become successors of v_{out}. The event count associated with v

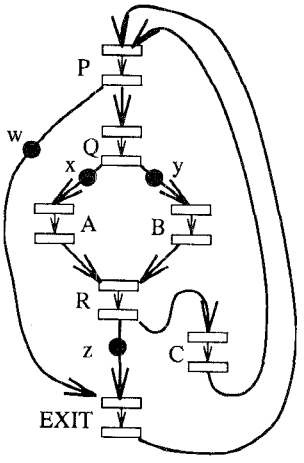


Fig. 2. Transformation of vertices to edges.

is then associated with the edge $v_{in} \rightarrow v_{out}$. Figure 2 shows the control-flow graph from Figure 1 after the above transformation has been applied to each vertex.

Each edge e contains a constant number of events, denoted by $Events(e)$. The goal of intraprocedural-event counting is to find a (small) set of edges $F \subseteq E$, and for each edge $f \in F$ a constant $Increment(f)$ such that for any directed cycle EX in the control-flow graph:

$$\sum_{\forall f \in F} \#(EX, f) Increment(f) = \sum_{\forall e \in E} \#(EX, e) Events(e).$$

As mentioned previously, some points in the execution path may not be accurate query points. We would like to find for each edge g, a query increment $QueryInc(g)$ such that for any *incomplete* execution path IX that starts at the *root* vertex and ends with edge g:

$$\left(\sum_{\forall f \in F} \#(IX, f) Increment(f) \right) + QueryInc(g) = \sum_{\forall e \in E} \#(IX, e) Events(e).$$

Section 3.1 shows that placing instrumentation code on the chords of any spanning tree of the control-flow graph is sufficient for event counting;[1] it also shows how to determine the increment for a chord from its fundamental cycle. Section 3.2 presents a depth-first search algorithm that computes the increment for all chords in linear time. Section 3.3 shows how to compute query increments.

[1] In practice, spanning trees must be chosen carefully in order to reduce the execution overhead incurred by instrumenting the chords. We will not concern ourselves with this issue here. As shown by Ball and Larus [1994], different spanning trees can produce very different run-time overhead.

3.1 Computing the Edge Event Increments

The increment value for a chord f is defined by its fundamental cycle $C(f)$ and the direction of the edges in this cycle:

$$Increment(f) = \sum_{\forall e \in C(f)} Events(e) Dir(C(f), e, f).$$

An edge e in $C(f)$ in the same direction as f makes a positive contribution to $Increment(f)$, while an edge e in the opposite direction from f makes a negative contribution. Of course, if $Increment(f) = 0$, then edge f does not have to be instrumented.

Figure 3 shows the fundamental cycle for each chord in the control-flow graph from Figure 2. The increment for chord w is $(+P + EXIT)$, since the edges P and $EXIT$ are directed in the same direction as edge w in $C(w)$. The fundamental cycle $C(z)$ defines $Increment(z) = (-C + EXIT)$, because edge C is in the opposite direction from z in $C(z)$ while edge $EXIT$ is in the same direction as z in $C(z)$. Note that edges P and R are not in $C(z)$, so they do not contribute to $Increment(z)$.

Using Theorem 2.1, it is straightforward to show that the above definition of $Increment(f)$ has the desired property, as shown by the following theorem:

THEOREM 3.1.1. *Let G be a control-flow graph, and let F be the set of chord edges induced by a spanning tree T for graph G (i.e., $F = E - T$). For any directed cycle EX in CFG G:*

$$\sum_{\forall f \in F} \#(EX, f) Increment(f) = \sum_{\forall e \in E} \#(EX, e) Events(e).$$

PROOF. Substituting the definition for $Increment(f)$ on the left-hand side of the above equation and moving $\#(EX, f)$ inside the inner sum of the resulting equation yields:

$$\sum_{\forall f \in F} \sum_{\forall e \in C(f)} \#(EX, f) Events(e) Dir(C(f), e, f).$$

The above equation is rewritten so that e is bound by the outer sum and f is bound by the inner sum, as follows:

$$= \sum_{\forall e \in E} \sum_{\substack{\forall f \in F \\ s.t.\ e \in C(f)}} \#(EX, f) Events(e) Dir(C(f), e, f).$$

Moving $Events(e)$ out of the inner sum yields:

$$= \sum_{\forall e \in E} Events(e) \sum_{\substack{\forall f \in F \\ s.t.\ e \in C(f)}} \#(EX, f) Dir(C(f), e, f).$$

By Knuth's theorem, the inner sum of the above equation is equal to $\#(EX, e)$, which finishes the proof. □

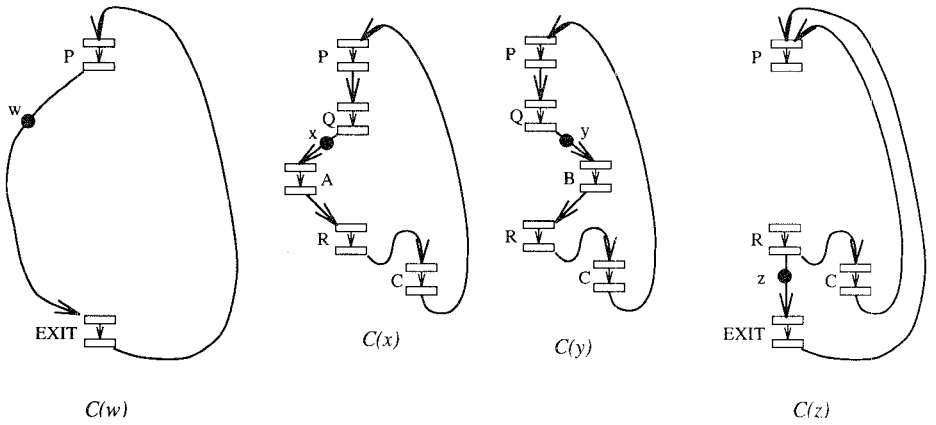


Fig. 3. The fundamental cycles of the graph from Figure 2 with respect to the spanning tree in that figure. The fundamental cycle $C(z)$ is shown with edges P and R hanging off it for context. Neither of these edges is in $C(z)$.

3.2 Computing the Edge Increments via a Depth-First Search

A naive algorithm for determining *Increment*(e) for each chord e would simply traverse each fundamental cycle $C(e)$. Since each fundamental cycle is, in the worst case, of size $O(E)$, and there are $E - (V - 1)$ chords, this naive algorithm would run in $O(E \times (E - V))$ worst-case time. Figure 4 presents an algorithm that uses a depth-first search of the spanning tree to determine the increments for all chords in time $O(E)$. There are three parameters to the depth-first search procedure DFS: an integer *events* which is the current event increment, a vertex v, and the tree edge e that was just traversed to reach v. Each chord is visited twice by the algorithm, while each tree edge is visited exactly once. The running time of this algorithm is clearly $O(E)$.

We explain the algorithm and show that it computes *Increment*(e) correctly for each chord e. At some point the depth-first search will have just traversed a path P that ends with chord e. For each edge f in P (except for e), *Events*(f) has been added to *events*. For each change in edge direction in P, *events* has been negated. For each edge f in P in the same direction as e, *Events*(f) will be negated an even number of times (through the negation of *events*), making a positive contribution to *Increment*(e) (when the assignment to *Increment*(e) takes place). For each edge f in P in the opposite direction from e, *Events*(f) will be negated an odd number of times, making a negative contribution to *Increment*(e). The event value associated with a chord is added to the chord's increment after the depth-first search.

To show that *Increment*(e) is correctly computed for any chord e, there are two cases to consider: the *root* vertex is in $C(e)$ or is not in $C(e)$. In the former case, the depth-first search will reach e two times by paths P_1 and P_2. The only edge shared by the two paths is e, and the union of edges in P_1 and P_2 is $C(e)$. As shown above, edges in the same direction as e make a

```
for each e ∈ E − T do Increment(e) := 0 od
DFS( 0, root, NULL )
for each e ∈ E − T do
  Increment(e) := Increment(e)+Events(e)
od

procedure DFS(integer events, vertex v, edge e)
  for each f ∈ T : f≠e and v=tgt(f) do
    DFS( Dir(e,f) *events + Events(f), src(f), f)
  od
  for each f ∈ T : f≠e and v=src(f) do
    DFS( Dir(e,f) *events + Events(f), tgt(f), f)
  od
  for each f ∈ E−T : v=src(f) or v=tgt(f) do
    Increment(f) := Increment(f) + Dir(e,f) *events
  fi

function Dir(edge e, edge f) : integer
  - precondition: e and f share a common endpoint
  if e = NULL then
    return (1)
  else if src(e)=tgt(f) or tgt(e)=src(f) then
    return (1)      // e and f in same direction
  else
    return (-1)     // e and f in opposite directions
  fi
```

Fig. 4. Algorithm for determining $Increment(e)$ for each chord e using a depth-first search of spanning tree T of graph G.

positive contribution to $Increment(e)$, while edges in the opposite direction from e make a negative contribution to $Increment(e)$, as desired.

Suppose that *root* vertex is not in $C(e)$. The depth-first search will still reach e two times by paths P_1 and P_2. However, in this case P_1 and P_2 share a common prefix Q containing edges that are not members of $C(e)$. Let f be the last edge in Q; let P_1' be the suffix of P_1 (i.e., $P_1 = Q \| P_1'$); and let P_2' be the suffix of P_2. The only edge shared by P_1' and P_2' is e, and the union of the edges in P_1' and P_2' is $C(e)$. It is clear that the event values for edges present in P_1' and P_2' will be correctly accounted for in $Increment(e)$. We must show that the event values accumulated in prefix Q cancel out. Suppose that edge f is encountered by the call DFS(x, v, f). It is clear that edge e cannot be in the same direction as edge f in both paths P_1 and P_2. So the event value x will be negated an even number of times along one path and an odd number of times along the other path, thus cancelling each other out.

3.3 Computing the Query Increments

As mentioned before, there is a trade-off between efficiently counting program events and the number of points in a program at which queries are accurate. If every program component updates the event counter, then queries are accurate at every component. Updating the event count at the chord edges of the spanning tree leads to fewer accurate query points. This problem is solved by computing a *query increment*, $QueryInc(g)$, for each edge g. If a query is made just after executing edge g, then $QueryInc(g)$ is simply added to the event counter to obtain the correct event count (the event counter is

not updated).[2] That is, for any *incomplete* execution path IX that starts at the *root* vertex and ends with edge g,

$$\sum_{\forall e \in E} \#(IX, e)Events(e) = \left(\sum_{\forall f \in F} \#(IX, f)Increment(f) \right) + QueryInc(g).$$

If $QueryInc(g) = 0$ then edge g is an accurate query point.

Computing $QueryInc(g)$ is easy. Let T be the set of spanning-tree edges. We add an edge $g' = tgt(g) \rightarrow root$ to the control-flow graph (with $Events(g') = 0$) and treat it as a member of $E - T$. Edge g' is a chord of the spanning tree T and has fundamental cycle $C(g')$. $QueryInc(g)$ is defined to be $Increment(g')$. The addition of edge g' to any incomplete path IX that starts at the *root* vertex and ends with g yields a directed cycle IX'. Since $Events(g') = 0$, it follows that

$$\sum_{\forall e \in E} \#(IX, e)Events(e) = \sum_{\forall e \in E} \#(IX', e)Events(e).$$

Since IX' is a directed cycle, Theorem 3.1.1 implies that

$$\sum_{\forall e \in E} \#(IX', e)Events(e) = \sum_{\forall f \in F} \#(IX', f)Increment(f).$$

As $Increment(g') = QueryInc(g)$ and IX' is formed by adding edge g' to IX, it follows that

$$\sum_{\forall f \in F} \#(IX', f)Increment(f) = \left(\sum_{\forall f \in F} \#(IX, f)Increment(f) \right) + QueryInc(g).$$

Figure 5 illustrates the computation of $QueryInc(B)$ for edge B from the control-flow graph from Figure 2. If the chord edge $B' = B_{out} \rightarrow P_{in}$ was added to this control-flow graph it would define the fundamental cycle $C(B')$ shown in Figure 5: $QueryInc(B) = Increment(B') = (-R - C)$.

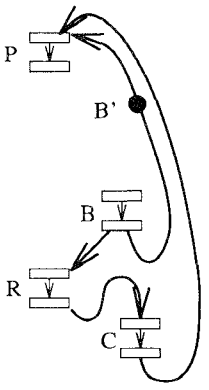
To compute the query increment for all edges in a control-flow graph, simply add a chord edge $e' = tgt(e) \rightarrow root$ for each original edge e and then apply the algorithm in Figure 4 to determine the increments for all chord edges.

4. INTERPROCEDURAL-EVENT COUNTING

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

The first problem is to maintain a global count of events executed throughout the entire program, with the capability to query the counter accurately at any place in the program. This problem is solved in two steps. The first step

[2] Query increments also can be used to update the event counter correctly when execution of a procedure terminates early (i.e., not at the *EXIT* vertex) due to an exceptional conditional or interprocedural transfer of control (such as setjmp / longjmp).



QueryInc(B) = Increment(B') = (-R -C)

Fig. 5. Computing *QueryInc*(*B*).

computes a spanning tree and determines the chord edge increments for each procedure's control-flow graph. The second step ensures that the event count is correct whenever a procedure passes control to another procedure by treating procedure calls as queries: immediately before a call the event counter is incremented by the query increment for the call, and immediately after the call the event counter is decremented by the same value (to ensure that counts will be correct after the call).[3] To query the event counter at vertex w in procedure X, we need only add *QueryInc*(w) to the event counter.

Without the second step, queries of the event counter would be prohibitively expensive, as the following example illustrates:

> Suppose that vertex B in the control-flow graph of Figure 1 contains a call to procedure X. When B passes control to X, the event count is off because *Increment*(y) has already accounted for vertices R and C, even though they have not executed. An accurate query from vertex w in procedure X must add both *QueryInc*(w) and *QueryInc*(B) to the event counter. In general, without the second step, the query may have to perform an addition for each active procedure.

The second problem is to maintain a count $cnt_{P,s}$ (for each call site 's: *call* Q' in procedure P) of events executed by the procedure Q (and procedures it calls transitively) when called from s in procedure P. A number of counters are used to solve this problem. For each procedure P, a global counter cnt_P is initialized to 0 at the beginning of program execution and tracks the events executed by P and procedures (transitively) called P. However, cnt_P is only updated inside the procedure P. For each (procedure P, call site 's: *call* Q') pair, there is a global counter $cnt_{P,s}$ and a counter $Lcnt_{P,s}$, which is a local variable of procedure P. The counter $cnt_{P,s}$ is initialized to 0 at the beginning of execution, and $Lcnt_{P,s}$ is initialized as described below.

Counter increments are determined as follows: For each procedure P, a spanning tree is found, and chord increments are computed. These incre-

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

ments are for events solely inside P and update the counter cnt_P. For each call site 's: *call* Q' in procedure P, the assignment '$Lcnt_{P,s} := cnt_Q$' is placed immediately before the call, and the assignments '$cnt_{P,s} := cnt_{P,s} + (cnt_Q - Lcnt_{P,s})$; $cnt_P := cnt_P + (cnt_Q - Lcnt_{P,s})$' are placed immediately after the call.

It is not difficult to see that this solution is correct by induction over the calling history of the program (as represented by a call tree). We show the base case. The induction step is quite similar. In the base case (at the leaves of the call tree) there is a call. By recording cnt_Q in $Lcnt_{s,P}$ before the call, the difference $(cnt_Q - Lcnt_{P,s})$ will be the number of events executed by procedure Q when called by P at call site s.

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

5. RELATED WORK

There are a number of works on the related topic of efficiently profiling programs with instrumentation [Knuth 1968; Knuth and Stevenson 1973; Probert 1975; Sarkar 1989; Samples 1991; Goldberg 1991; Ball and Larus 1994]. All of these (except Sarkar [1989]) use the spanning tree to determine a (small) set of points in a control-flow graph at which to place counters so that vertex (basic block) profiles or edge profiles can be derived from the measured points. Knuth's theorem, stated in Section 2, shows that the number of times each control-flow edge appears in an execution is uniquely determined by the number of times each chord (of some spanning tree) appears in the execution. In edge profiling, each chord has an associated counter that is incremented each time the chord executes. After execution, Knuth's theorem can be applied to determine the count for tree edges from the chords' counts. In event counting, we have made use of Knuth's theorem (before program execution) to determine an increment value for each chord that summarizes the event counts associated with tree edges. A chord's increment value may be zero, in which case instrumentation of that chord is unnecessary (unlike in profiling, in which each chord edge must be instrumented).

Mellor-Crummey and LeBlanc [1989] describe what they call a *software instruction counter*. The software instruction counter does not actually count the number of instructions that have executed. Rather, the software instruction counter is a pair (PC, SIC) where PC is the program counter and SIC is a counter that is incremented for each backward branch and procedure call that has executed. Because a program counter's value can only be reused if a backward branch is taken or a chain of recursive calls is made, this pair of values uniquely identifies a particular instruction in a program's execution

history. Such a counter has utility in cyclic debugging where one is interested in repeated executions and stopping at a particular state. However, as defined, the software instruction counter cannot count the number of instructions executed.

6. CONCLUSION

Events in a program's execution may be counted efficiently by instrumenting the chords of a spanning tree of the control-flow graph. An efficient depth-first search algorithm computes the increment value for each chord and a query increment for each vertex at which a query needs to be made.

REFERENCES

BALL, T. AND LARUS, J. R. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst. 16*, 3 (July).

GOLDBERG, A. 1991. Reducing overhead in counter-based execution profiling. Tech. Rep. CSL-TR-91-495, Computer Systems Laboratory, Stanford Univ., Stanford, Calif., (Oct.).

GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. 1983. An execution profiler for modular programs. *Softw. Pract. Exper. 13*, 671–685.

HOLLINGSWORTH, J. K. AND MILLER, B. P. 1993. Dynamic control of performance monitoring on large scale parallel systems. Tech. Rep. #1133, Univ. of Wisconsin, Madison (Jan.).

KESSLER, P. B. 1990. Fast breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. SIGPLAN Not. 25* 6 (June), 78–84.

KNUTH, D. E. 1973. *The Art of Computer Programming*, Vol. 1. *Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, Mass.

KNUTH, D. E. AND STEVENSON, F. R. 1973. Optimal measurement points for program frequency counts. *BIT 13*, 313–322.

MELLOR-CRUMMEY, J. M. AND LEBLANC, T. J. 1989. A software instruction counter. In the *3rd ASPLOS Proceedings. SIGARCH Comput. Arch. News 17*, 2, 78–86.

PROBERT, R. L. 1975. Optimal insertion of software probes in well-delimited programs. *IEEE Trans. Softw. Eng. SE-8*, 1 (Jan.), 34–42.

REINHARDT, S. K., HILL, M. D., LARUS, J. R., LEBECK, A. R., LEWIS, J. C., AND WOOD, D. A. 1993. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM, New York, 48–60.

SAMPLES, A. D. 1991. Profile-driven compilation. Ph.D. thesis, Rep. No. UCB/CSD 91/627, Univ. of California, Berkeley, Calif., Apr.

SARKAR, V. 1989. Determining average program execution times and their variance. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (Portland, Ore). ACM SIGPLAN Not. 24*, 7, 298–312.

Received October 1993; revised March 1994; accepted April 1994