Efficient Path Profiling

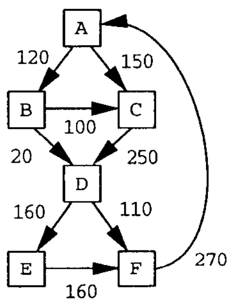
Thomas Ball
Bell Laboratories
Lucent Technologies
tball@research.bell-labs.com

James R. Larus*
Dept. of Computer Sciences
University of Wisconsin-Madison
larus@cs.wisc.edu

Abstract

A path profile *determines how many times each acyclic path in a routine executes. This type of profiling subsumes the more common basic block and edge profiling, which only approximate path frequencies. Path profiles have many potential uses in program performance tuning, profile-directed compilation, and software test coverage.*

This paper describes a new algorithm for path profiling. This simple, fast algorithm selects and places profile instrumentation to minimize run-time overhead. Instrumented programs run with overhead comparable to the best previous profiling techniques. On the SPEC95 benchmarks, path profiling overhead averaged 31%, as compared to 16% for efficient edge profiling. Path profiling also identifies longer paths than a previous technique, which predicted paths from edge profiles (average of 88, versus 34 instructions). Moreover, profiling shows that the SPEC95 train *input datasets covered most of the paths executed in the* ref *datasets.*



Path	Prof1	Prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20

Figure 1. Example in which edge profiling does not identify the most frequently executed paths. The table contains two different path profiles. Both path profiles induce the same edge execution frequencies, shown by the edge frequencies in the control-flow graph. In path profile $Prof1$, path $ABCDEF$ is most frequently executed, although the heuristic of following edges with the highest frequency identifies path $ACDEF$ as the most frequent.

1 Introduction

Program profiling counts occurrences of an event during a program's execution. Typically, the measured event is the execution of a local portion of a program, such as a routine or line of code. Recently, fine-grain profiles—of basic blocks and control-flow edges—have become the basis for profile-driven compilation, which uses measured frequencies to guide compilation and optimization.

One use of profile information is to identify heavily executed paths (or traces) in a program [Fis81, Ell85, Cha88, YS94]. Unfortunately, basic block and edge profiles, although inexpensive and widely available, do not always correctly predict frequencies of overlapping paths. Consider, for example, the control-flow graph (CFG) in Figure 1. Each edge in the CFG is labeled with its frequency, which normally results from dynamic profiling, but in the figure is induced by *both* path profiles in the table. A commonly used heuristic to select a heavily executed path follows the most frequently executed edge out of a basic block [Cha88], which identifies path $ACDEF$. However, in path profile $Prof1$, this path executed only 60 times, as compared to 90 times for path $ACDF$ and 100 times for path $ABCDEF$. In profile $Prof2$, the disparity is even greater although the edge profile is exactly the same.

This inaccuracy is usually ignored, under the assumption that accurate path profiling must be far more expensive than basic block or edge profiling. Path profiling is the ultimate form of control-flow profiling, as it uniquely deter-

*This research supported by: Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550; NSF NYI Award CCR-9357779, with support from Hewlett Packard, Sun Microsystems, and PGI; NSF Grant MIP-9225097; and DOE Grant DE-FG02-93ER25176.

mines both basic block and edge profiles, although the converse does not hold, as Figure 1 shows. Also, the number of blocks or edges in a program is finite and linear in the program's size, but a program with loops offers an unbounded number of potential paths. Considering only acyclic paths bounds this set, but, in the worst case, its size is still exponential in the program's size.

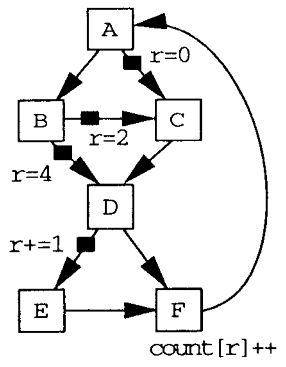
This paper shows that accurate profiling is neither complex nor expensive. It describes a new and efficient technique for *path profiling*. Our algorithm places instrumentation that accurately determines dynamic execution frequency of control-flow paths in a routine. The instrumentation is not only simple and low-cost, but it is placed in a way that minimizes its overhead. Remarkably, although path profiling collects far more information than block or edge profiling, its overhead can be lower and is usually comparable—on the SPEC95 benchmarks, path profiling's average overhead is 31%, while efficient edge profiling's overhead is 16%.

Efficient path profiling opens new possibilities for program optimization and performance tuning. Instead of relying on heuristics, which fully predict only 38% of the executed acyclic paths in the SPEC95 benchmarks, profile-driven compilers can base their decisions on accurate measurements.

Another potential application of path profiling is software test coverage, which quantifies the adequacy of a test data set by profiling a program and reporting unexecuted statements or control-flow. Few, if any, coverage tools measure path coverage. Instead, tools rely on weaker criteria, such as statement or control-flow edge coverage. Edge profiling is less complete than path profiling, as shown in Figure 1, where the two path profiles cover different sets of paths yet induce the same edge profile. Besides an efficient algorithm for path profiling, this paper also presents measurements that show that most *routines* in a small sample of programs have few (< 3000) potential paths, so that path coverage testing could be feasible for large portions of a program. On the other hand, the measurements also demonstrate the difficulty of developing test data sets, since the programs *as a whole* executed an average of 2696 paths (249–24414), as compared to the millions of potential paths identified by the path profiling algorithm.

1.1 Algorithm Overview

The essential idea behind the path profiling algorithm is to identify sets of potential paths with states, which are encoded as integers. Consider for a moment a routine without a loop. Upon entry to the routine, all paths are possible. Taking a conditional branch narrows the set of potential paths and corresponds to a transition to a new state. At the routine's exit, the final state represents the single path taken



Path	Encoding
ACDF	0
ACDEF	1
ABCDF	2
ABCDEF	3
ABDF	4
ABDEF	5

Figure 2. Path profiling instrumentation. Each path from A to F produces a unique state in register r, which indexes an array of counters in F.

through the routine. This paper presents an efficient algorithm that:

- Numbers final states from $0 \ldots n - 1$, where n is the number of potential paths in a routine. With this compact numbering, a final state can directly index an array of counters.
- Places instrumentation so that transitions need not occur at every conditional branch.
- Assigns states so that transitions can be computed by a simple arithmetic operation, without an explicit state transition table or memory reference.
- Transforms a control-flow graph containing loops or huge numbers of potential paths into an acyclic graph with a limited number of paths.

Figure 2 illustrates the technique. Edges labeled by small squares contain instrumentation, which updates the state in register r. The loop contains six unique paths, and each one computes a different value for r, as shown in the table. At the end of the loop body (block F), register r holds the index to increment an array of counters.

1.2 Extensions

The algorithm in this paper can be easily extended in several ways. First, instead of intraprocedural profiling, it could be applied to a program's call graph, to record call paths. An interesting complication is indirect calls, which require a dynamic data structure to record calls along edges that are not in the call graph.

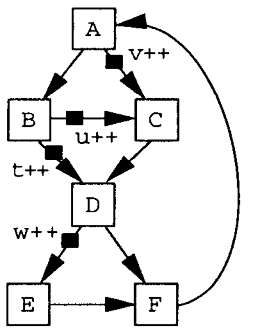
Also, instead of just counting the number of times a path executes, the profiling algorithm can easily accumulate a metric for a path. Some processors provide accessible counters for metrics such as the number of processor cycles, stalls, cache misses, or page faults. A minor change to the path profiling code could increment a path's counter by the change in a counter over the path.

1.3 Paper Overview

The path profiling algorithm instruments a program to record paths with low run-time overhead. The algorithm uses previous results on efficient profiling and tracing [BL94] and efficient event counting [Bal94] to determine which edges to instrument. The contribution of this paper is combine these algorithms, apply them to a new problem, and develop a new algorithm to compute an update constant for each instrumented edge. The algorithm ensures that each distinct path generates a unique value. Furthermore, the path encoding is compact and minimal, so that the maximum value for any path is the number of unique paths through a CFG (minus one), as in Figure 2. A simple, linear-time algorithm achieves both goals (Section 3).

This paper only considers intra-procedural acyclic paths, which result from removing loop backedges before instrumentation (Section 4). This process produces a profile that does not capture paths that cross a backedge. However, acyclic path profiling counts the number of times that a loop iterates and records both paths into the first iteration and out of the last loop iteration. The same approach, of removing edges, can also limit the number of paths in complex routines, so that states can be represented as 32-bit integers. Even so, large routines can have too many states to use an array of counters. In this case, a hash table records paths that actually execute, so that the space overhead is proportional to the number of dynamic paths, rather than the number of potential static paths. The relatively high cost of hashing accounts for the higher overhead of path profiling, as compared to edge profiling.

We implemented the algorithms presented here in a profiling tool, *PP*, which uses the EEL library [LS95] to insert instrumentation into executable binaries (Section 5). This paper compares PP against QPT2, another profiling tool built with EEL, which uses an efficient edge profiling algorithm [BL94]. QPT2 usually incurred less overhead, but the two system were roughly comparable. Profiling the SPEC95 benchmarks, PP's overhead averaged 31% (6–97%) while QPT2's overhead averaged 16% (-2.6–53%) (Section 6). The measurements also compare profiled paths against paths predicted using edge profiles and show that for the SPEC95 benchmarks, which execute few unique paths, profiling identifies longer paths (an average of 7 CFG edges and 88 instructions, versus 5 edges and 34 instructions for predicted paths). Moreover, path profiling shows that the paths executed with the SPEC95 *train* dataset cover most of the dynamically executed instructions in the *ref* dataset, which suggests that path profiles could help improve SPEC95 peak numbers.



C->D	=	u + v
D->F	=	t + u + v - w
E->F	=	w
A->B	=	t + u
F->A	=	t + u + v

Figure 3. Instrumentation for edge profiling.

2 Related Work

The path profiling algorithm, like previous work on efficient profiling and tracing techniques [BL94, Gol91], uses a spanning tree to determine a minimal, low-cost set of edges to instrument. For example, Figure 3 shows the control-flow graph from Figure 2 instrumented for edge profiling (the uninstrumented edges form a spanning tree). The same set of edges are instrumented in both cases. However, for edge profiling, each instrumented edge has its own counter (held in memory), which is incremented each time the edge executes. Figure 3 also shows how uninstrumented edges' counts are derived from recorded counts.

Path profiling produces a more detailed profile, although it instruments the same set of edges. Moreover, most path profiling instrumentation consists of register instructions, while every edge profiling instrumentation increments memory. In general, a path that executes N memory increments for edge profiling will execute N register initializations/adds plus one memory increment for path profiling. In practice, there are many procedures for which the number of potential paths is small (so arrays may be used) and path profiling incurs less overhead than edge profiling. However, there are procedures that have so many potential paths that a hash table must be used to store the profile.

Young and Smith used a limited form of program tracing to record paths for their branch correlation studies [YS94]. In a FIFO buffer, they recorded the last n branches, each of which consists of a basic block number and branch outcome. This technique is both more expensive than path profiling and also requires another level of indirection to associate a counter with a path, which consists of a sequence of block numbers. Unlike path profiling, this technique need not distinguish cyclic from acyclic paths since it truncates both at the FIFO boundary.

Bit tracing is another approach to path profiling. Bit tracing associates a 1-bit value with the outcome of each two-

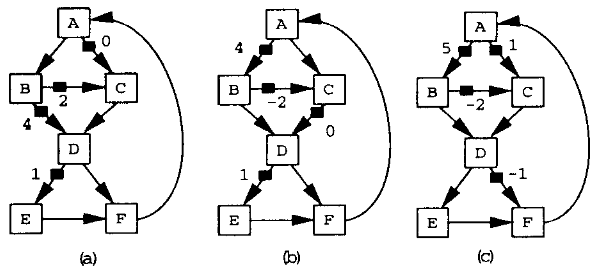


Figure 4. Three possible placements of instrumentation for the control-flow graph from Figure 1.

way branch [BL94, Bal96]. When a branch executes, instrumentation code appends a bit to a trace buffer that records branch outcomes. By recording multiple bits, the approach can be extended to multi-way branches. The contents of the buffer form an index into an array or as a hash value.

It is easy to see that bit tracing uses the minimal number of bits necessary to distinguish paths. For simple control-flow graphs, such as a chain of if-then-else statements, bit tracing, like our approach, produces a compact representations of paths. However, in general, bit tracing may not yield the most compact representations of paths possible. It is easy to construct examples for which the maximal path value under bit tracing is not minimal, no matter the choice of bit labellings. In the worst case, the number of entries in an array of counters may be twice our method.

In addition, bit tracing is likely to have higher run-time overhead than our approach. First, every predicate must be instrumented, whereas our approach allows flexibility in placing instrumentation to reduce overhead. Second, on most machines, the instrumentation to append to a bit string is more complex and slower than a register-to-register addition.

3 Path Profiling of DAGs

As described previously, path profiling tracks a path in a directed acyclic graph (DAG) by updating a register along certain edges of the DAG. This section shows how to compute the necessary updates, efficiently place instrumentation, and derive an executed path from the resulting profile.

The example in Figure 4 shows that many placements of instrumentation yield equivalent results. However, some placements incur less run-time overhead than others. For example, all three graphs in Figure 4 produce the same sum along any acyclic path from A to F. However, in graph (a), the largest number of instrumented edges on any path from A to F is two, while graphs (b) and (c) have up to four and three, respectively.

The path profiling algorithm first labels edges in a DAG with integer values, such that each path from the entry to the exit of the DAG produces a unique sum of the edge values along that path (the *path sum*). However, placements from this step may have sub-optimal run-time overhead, as above.

In the next step, another algorithm [Bal94] improves this computation, by finding an equivalent computation that uses a minimal number of additions along DAG edges that are not in the DAG's spanning tree. In each graph in Figure 4, the uninstrumented edges (those without squares along them) form a spanning tree. Since a DAG may have many spanning trees, the algorithm has the freedom to place instrumentation along edges less likely to be executed.[1]

After reviewing the basic graph terminology in Section 3.1, this section describes the four basic steps to path profile a DAG:

1. Assign integer values to edges such that no two paths compute the same path sum (Section 3.2). This encoding is minimal.
2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge (Section 3.3).
3. Select appropriate instrumentation (Section 3.4).
4. After collecting the run-time profile, derive the executed paths (Section 3.5).

3.1 Terminology

For the remainder of this paper, unless otherwise noted, control-flow graphs (CFGs) have been converted into directed acyclic graphs (DAG) with a unique source vertex $ENTRY$ and sink vertex $EXIT$. Section 4 shows how to transform an arbitrary CFG into a DAG, which can be path profiled. For technical reasons, the increment computation (Section 3.3) requires a "dummy" edge $EXIT \rightarrow ENTRY$ (although this creates an unexecutable cycle, the graph can still be treated as a DAG by ignoring this backedge).

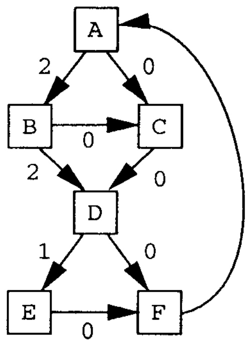
An execution of a DAG produces an acyclic, directed path starting at $ENTRY$ and ending at $EXIT$. The term *path* refers to an acyclic directed path, unless otherwise noted. Of course, a DAG may execute many times, as it may consist of a loop body or a procedure.

A spanning tree of a graph G is a subgraph that is a tree and contains all vertices of G. Edges in a spanning tree are bidirectional and need not follow the direction of graph

[1]This approach requires computing (or obtaining from a profile) a weight for each edge that statically approximates the edge's execution frequency. A maximum spanning tree of the graph, with respect to that weighting, maximizes the weight (execution frequency) of the uninstrumented edges. PP uses the same previously published, effective algorithm for statically computing a weighting as QPT [BL94].

```
foreach vertex v in reverse topological order {
  if v is a leaf vertex {
    NumPaths(v) = 1;
  } else {
    NumPaths(v) = 0;
    for each edge e = v->w {
      Val(e) = NumPaths(v);
      NumPaths(v) = NumPaths(v) + NumPaths(w);
    }
  }
}
```

Figure 5. Algorithm for assigning values to edges in a DAG.



Vertex v	NumPaths(v)
A	6
B	4
C	2
D	2
E	1
F	1

Figure 6. Control-flow graph from Figure 1, with values computed by the algorithm in Figure 5.

edges. If T is the set of spanning tree edges, then any graph edge not in T is a *chord* of the spanning tree.

For example, in the graph of Figure 2, vertex A is the $ENTRY$ vertex and vertex F is the $EXIT$ vertex. The unadorned graph edges comprise a spanning tree. The edges labeled by squares are chords of the spanning tree.

3.2 Compactly Representing Paths with Sums

The first step in path profiling is to assign a non-negative constant value $Val(e)$ to each edge e in a DAG, such that the sum of values along any path from $ENTRY$ to $EXIT$ is unique. Furthermore, the path sums should lie in the range from 0 to the number of paths (minus one), so that the encoding is minimal.

The algorithm in Figure 5 computes such a Val relation by visiting vertices of the DAG in reverse topological order. This order ensures that all the successors of a vertex v are visited before v itself. Associated with each vertex v is a value $NumPaths(v)$, which records the number of paths from v to $EXIT$. The algorithm is simple. At vertex v, the algorithm visits all of v's outgoing edges $v \to w_i$, $1 \le i \le n$, and assigns the k^{th} outgoing edge the value:

$$Val(v \to w_k) = \sum_{i=1}^{k-1} NumPaths(w_i)$$

The following theorem proves the algorithm correct:

Theorem 1 *Given a DAG, after the algorithm of Figure 5 visits vertex v, $NumPaths(v)$ is the number of paths from v to $EXIT$ and each path from v to $EXIT$ generates a unique value sum in the range* $0 \ldots NumPaths(v) - 1$.

Proof. By induction on the height of a vertex in the DAG (i.e., the max number of steps to the sink vertex $EXIT$).

Base Case: v has height equal to zero (that is, $v = EXIT$), so $NumPaths(v) = 1$. The theorem is trivially satisfied.

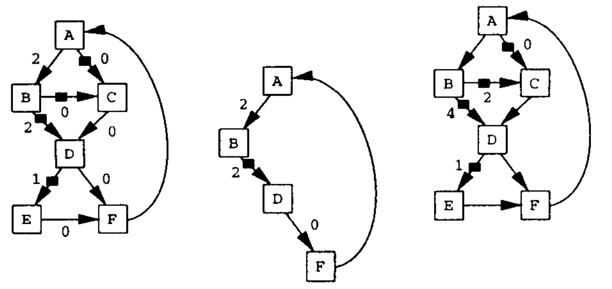
Induction Step: Show that the theorem holds for any vertex v of height H ($H > 0$). All successors $w_1 \ldots w_n$ of v must have height less than H (because the graph is a DAG), so the theorem holds for all w_i. It is trivial to see that the number of paths from v to $EXIT$ is $\sum_{i=1}^{n} NumPaths(w_i)$, which the algorithm computes. By the induction hypothesis, each path from w_k to $EXIT$ generates a unique value sum in the range $0 \ldots NumPaths(w_k) - 1$. Therefore, any path from v to $EXIT$ starting with edge $v \to w_k$ will generate a unique value in the range $\sum_{i=1}^{k-1} NumPaths(w_i) \ldots (\sum_{i=1}^{k} NumPaths(w_i)) - 1$. Since all $NumPaths(w_i)$ values are greater than 0, it follows that no two paths from v to $EXIT$ generate the same value sum. □

Figure 6 illustrates how the algorithm operates on the example control-flow graph. Note that vertices are labeled in topological ordering, so $FEDCBA$ is a reverse topological order. Any vertex with a single outgoing edge e, such as C and E, always has $Val(e) = 0$.

3.3 Efficiently Computing Sums

Given an edge value assignment, the second step of the algorithm finds a minimal cost set—with respect to a weighting (Section 3)—of edges along which to compute these values, while preserving the two properties of the value assignment.

This step of the algorithm finds a maximal cost spanning tree of the graph (to find a minimal cost set of chord edges), and applies an efficient event counting technique [Bal94] to determine the increment $Inc(c)$ for each chord c in a spanning tree. The event counting algorithm ensures that the sum of Inc values for any path P from $ENTRY$ to $EXIT$ is identical to the sum of Val values for P. Note that some of the Inc values may be negative, as in Figure 4. The edge $EXIT \to ENTRY$ is required for this step (if this edge is



```
Inc(B->D)  = Val(A->B) + Val(B->D) + Val(D->F) + Val(F->A)
           = 2 + 2 + 0 + 0
           = 4
```

Figure 7. Application of event counting algorithm to determine chord increments.

selected as a chord, then its instrumentation can be placed in the *EXIT* vertex).

Figure 7 shows how the event counting algorithm applies to the example control-flow graph. The graph on the left contains the value assignment and the chord edges (those with squares along them). The graph in the middle shows the unique cycle of spanning tree edges associated with chord $B \rightarrow D$. The values along edges in this cycle determine the increment for chord $B \rightarrow D$, which in this case is four. Informally, the algorithm propagates the value of two from edge $A \rightarrow B$ to the chord $B \rightarrow D$. The graph on the right contains increments for all chords. Each path from A to F in this graph yields the same path sum as in the graph on the left.

3.4 Instrumentation

After computing chord increments, the algorithm selects instrumentation. Of course, at the start of a program's execution, the array of counters must be allocated and initialized to 0. At program termination, this array is written to permanent storage.

Besides this prelude and postlude instrumentation, the remaining instrumentation has three tasks: initializing path register r [`r = 0`] in the *ENTRY* vertex; updating r in chord c [`r += Inc(c)`]; and incrementing a path's memory counter in the *EXIT* vertex [`count[r]++`]. However, in many cases, an optimization can combine updates with the other two operations, as shown in Figure 2.

The optimization for initialization is:

> (1) A chord c may initialize the path register [`r=Inc(c)`] if and only if c is the first chord in every path from *ENTRY* to *EXIT* containing c.

If there exists a path in which chord c is the first chord and another path in which c is not the first chord, then chord c must update rather than initialize r. However, in this case, moving the initialization [`r=0`] as close to c as possible avoids redundant initialization of r.

A similar optimization works for counter increments:

> (2) A chord c may increment the path register and memory counter [`count[r+Inc(c)]++`] if and only if c is the last chord in every path from *ENTRY* to *EXIT* containing c.

This optimization can fold an addition into a memory address calculation.

If a chord c contains initialization as well as a counter increment, the instrumentation simply becomes `count[Inc(c)]++`.

The algorithm in Figure 8 places instrumentation properly. The first **while** loop moves initialization code to chord edges when possible, and otherwise moves it far enough from the *ENTRY* vertex so that no initialization is redundant. The second loop places the memory increment code. The invariant of the first loop is that for each vertex w added to the working set, there is only one path from *ENTRY* to w in the DAG and this path contains no chords. Note that if there are two paths from *ENTRY* to a vertex w, one of these paths must contain a chord, so any chord encountered from w onward cannot satisfy this condition. A similar invariant is maintained by the second **while** loop.

Figure 9 shows the instrumentation for the control-flow graph from Figure 1. Note that edges $A \rightarrow C$, $B \rightarrow C$ and $B \rightarrow D$ now initialize register r, eliminating the need to initialize r at vertex A. Furthermore, the update along edge $D \rightarrow E$ has been combined with the counter increment code. However, notice that a counter increment is required along edge $D \rightarrow E$ as well.

3.5 Regenerating a Path

To recreate a path profile from the path counters recorded at run time, it is necessary to map from the integer representing a path to the path itself. This is done using the value assignment computed previously (Section 3.2).

The regeneration algorithm is straight forward. Regeneration starts from a control flow graph's *ENTRY* node and traverses the graph, using the path value to select which edge to follow out of a basic block. Let v be a vertex in the reconstructed path and let R be the path value. Initially, $v = ENTRY$ and R is the number of the path to regenerate. At each block, find $e = v \rightarrow w$, which is the outgoing edge of v with the largest $Val(e) \leq R$. As the path traverses edge e, let $v = w$ and $R = R - Val(e)$. Repeat this process until control reaches the *EXIT* vertex.

For example, consider the control-flow graph in Figure 6. Suppose that the initial path value R is 3. At vertex A, the algorithm will choose edge $A \rightarrow B$ and decrement R by 2.

```
// Register initialization code
//
WS.add(ENTRY);
while not WS.empty() {
  vertex v = WS.remove();
  for each edge e = v->w
    if e is a chord edge
      instrument(e, 'r=Inc(e)');
    else if e is the only incoming edge of w
      WS.add(w);
    else instrument(e, 'r=0');
}

// Memory increment code
//
WS.add(EXIT)
while not WS.empty() {
  vertex w = WS.remove();
  for each edge e = v->w
    if e is a chord edge {
      if e's instrumentation is 'r=Inc(e)'
        instrument(e, 'count[Inc(e)]++');
      else
        instrument(e, 'count[r+Inc(e)]++');
    } else if e is the only outgoing edge of v
      WS.add(v);
    else instrument(e, 'count[r]++');
}

// Register increment code
//
for all uninstrumented chords c
  instrument(c,'r+=Inc(c)')
```

Figure 8. Algorithm for placing instrumentation.

At vertex B, $R = 1$, so the algorithm traverses edge $B \rightarrow C$ and then $C \rightarrow D$. At vertex D, R still has a value of 1, so the path traverses edge $D \rightarrow E$, followed by $E \rightarrow F$. The resulting regenerated path is $ABCDEF$, which is the path that generates the path sum 3.

3.6 Early Termination

Like other efficient profiling algorithms [BL94], path profiling requires extra information to derive correct profiles for routines that terminate unexpectedly because of exceptions, unrecognized non-local gotos, or calls to exit. This information consists of the address of unterminated calls and can easily be obtained from a program's stack at an unexpected event. The event counting algorithm provides a way to correctly update the counters in these routines [Bal94].

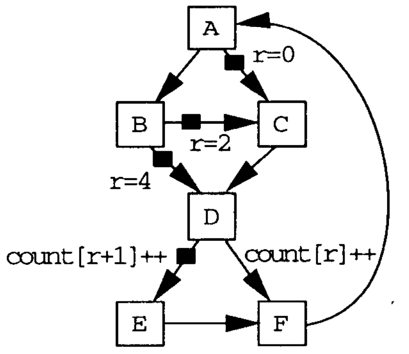


Figure 9. Optimization of instrumentation for the control-flow graph of Figure 1.

4 Path Profiling of Arbitrary Control-Flow

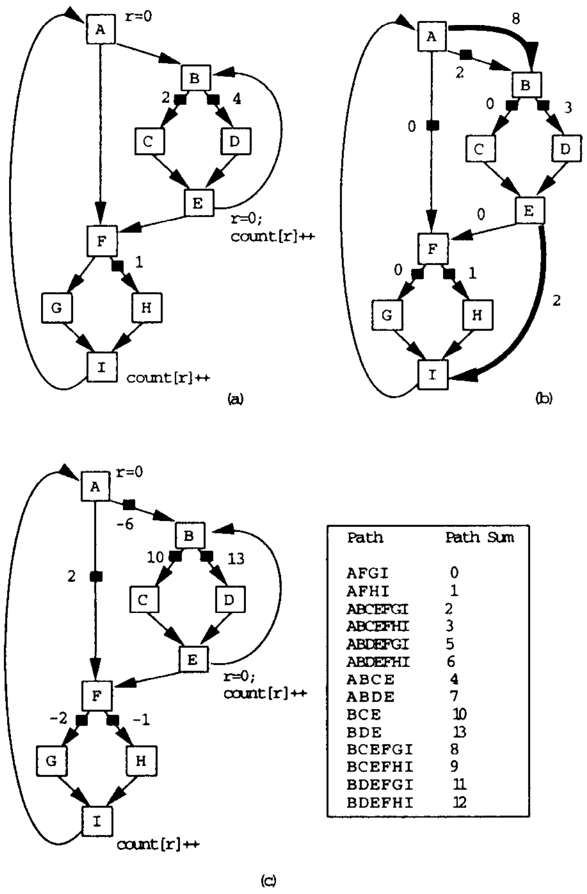
This section extends path profiling to arbitrary control-flow graphs that contain cycles (including irreducible loops). Any cycle in a control-flow graph must contain a backedge (as identified by a depth-first search of the graph). The algorithm in Section 3 only works for acyclic paths, which correspond to backedge-free paths.

Our approach to handling general CFGs instruments each backedge with a path counter increment and path register initialization [`count[r]++; r = 0`], which records the path up to the backedge and prepares to record the path after the backedge.

Suppose that $v \rightarrow w$ and $x \rightarrow y$ are backedges. A general CFG contains four possible types of acyclic (backedge-free) paths:

- A path from $ENTRY$ to $EXIT$.
- A path from $ENTRY$ to v, ending with execution of backedge $v \rightarrow w$.
- A path from w to x (after execution of backedge $v \rightarrow w$), ending with execution of backedge $x \rightarrow y$ (note: $v \rightarrow w$ and $x \rightarrow y$ may be the same edge).
- After executing backedge $v \rightarrow w$, a path from w to $EXIT$.

Removing all backedges from a control-flow graph produces a DAG (as defined in Section 3.1). However, simply applying the profiling algorithm from Section 3 to this DAG will not correctly distinguish the above four types of paths. Figure 10(a) contains a control-flow graph with a loop consisting of the vertices B, C, D, and E. Suppose the graph is instrumented by eliminating the backedge $E \rightarrow B$, thus yielding a DAG, and applying the path profiling algorithm for DAGs. The resulting assignment does not ensure that different paths yield different paths sums. For example, the



Path	Path Sum
AFGI	0
AFHI	1
ABCEFGI	2
ABCEFHI	3
ABDEFGI	5
ABDEFHI	6
ABCE	4
ABDE	7
BCE	10
BDE	13
BCEFGI	8
BCEFHI	9
BDEFGI	11
BDEFHI	12

Figure 10. Control-flow graph with a loop.

paths BCE, $ABCE$, $BCEFGI$ and $ABCEFGI$ all compute the identical path sum of 2. Not surprisingly, only paths that start at $ENTRY = A$ and end at $EXIT = I$ are correctly distinguished.

The solution to this problem consists of three steps:

- For each vertex v that is the target of one or more backedges, add a dummy edge $ENTRY \dashrightarrow v$. For each vertex w that is the source of one (or more) backedges, add a dummy edge $w \dashrightarrow EXIT$. If one of these edges is not in the spanning tree, it will be instrumented, which is efficient as the edge's increment can be combined with the code always added to a loop backedge to record a path.

- Eliminate backedges from the graph (except for the edge $EXIT \dashrightarrow ENTRY$, which was added for increment computation).

- Apply the first two steps of the path profiling algorithm (Section 3) to compute a value assignment and chord increments.

The dummy edges create extra paths from $ENTRY$ to $EXIT$, which the value assignment algorithm takes into account. The dummy edge from $ENTRY$ to a loop head corresponds to reinitializing the path register along the loop's backedge. The dummy edge from the loop's bottom to $EXIT$ corresponds to incrementing the path counter along the backedge.

Figure 10(b) shows the graph after this transformation and edge value assignment. Dummy edges are the thicker edges. As a result, the chord increments correctly distinguish the four classes of paths listed above. Figure 10(c), shows the chord increments computed and the path sum for each possible path through the graph.

Path regeneration must follow the first two steps (adding dummy edges and removing backedges) to compute the same value assignment, before using the regeneration algorithm from Section 3.5 on the resulting graph.

4.1 Self Loops

The approach described must be slightly modified to handle self-loop edges, which are backedges with the same source and target vertex. Removing this edge does not leave any edge in the loop to instrument. These edges can be handled specially, by adding a counter along them to record the number of times they execute, rather than instrumenting them with the code [`count[r]++; r = 0`].

5 Implementation

We implemented the algorithms described previously in a tool called PP, which instruments SPARC binary executables to extract path profiles. PP is built on EEL (Executable Editing Library), which is a C++ library that hides much of the complexity and system-specific detail of editing executables [LS95]. EEL provides abstractions that allow a tool to analyze and modify binary executables without being concerned with particular instruction sets, executable file formats, or the consequences of deleting existing code and adding foreign code (i.e., instrumentation).

5.1 Registers

Path profiling requires a local register throughout each routine's execution to hold the current path and a temporary register for some instrumentation code, such as the memory increment code. EEL scavenges free registers by using dataflow analysis to find the dead registers throughout a control-flow graph. If EEL cannot find an unused local register, it frees the least heavily used local register by spilling it to the routine's stack frame. In most routines, EEL found unused local registers, although many larger and computationally intensive routines require spill code. The SPARC's register windows ensure that all local registers are caller saved. Other architectures would need to save the path register before and after calls.

EEL also provides a facility to add procedure calls at arbitrary points in a routine. PP uses this feature, which relies on program analysis to save only live values, to call the hashing code (see Section 5.3).

5.2 Optimizations

A simple strength-reduction optimization saves two instructions per path by having the path register hold a counter's address, instead of its index. PP initializes a path register to the base of the counter array. Each increment adds its update, scaled by the size of a counter (4 bytes). This optimization saves three instructions in the code that increments a path's counter, at the cost of an additional instruction in the code that initializes the path register. Unfortunately, the optimization reduces the range of increments that fit in an instruction's immediate field. Since the SPARC's immediate field is 13 bits, this optimization is limited to routines in which the largest increment is 1023 (rather than 4095). However, most routines have fewer or far more paths.

Moreover, a simple change can reduce the range of increments in a routine. The algorithm in Figure 5 can visit a node's successors in any order. By visiting the successor with the largest number of paths (`NumPath`) last, the value (`Val`) assigned to the last edge is minimized, and hence so are the increments added at run time.

5.3 Routines with Many Paths

PP employs two techniques to handle routines with a large number of paths. The first, which PP applies to any routine in which an increment is larger than an instruction's immediate field, replaces the array of counters with a hash table. On the SPARC, this means that routines with more than 4000–6000 paths require hash tables. Hash tables have the advantage of requiring space proportional to the number of executed paths, but have the disadvantage of being an order of magnitude more costly than a simple memory increment and forcing a function call in awkward places. This technique limits counter space for a routine to roughly 16K bytes (most routines require far less space). PP uses two hash routines. The one called from loop backedges keeps a pointer to the last path and hash bucket, so that repeated lookups of the same path are very fast. This is very beneficial since procedures often spend their time in tight loops, repeating the same path over and over.

PP's other technique is necessary for very complex routines in which the number of possible paths exceeded the range of a 32 bit integer. In these routines, PP terminates the value computation (Section 3.2) when the number of paths reachable from a node exceeded a threshold (currently 100,000,000). At this point, PP removes all outgoing edges from the node—using the same approach to terminate these paths as for loop backedges (Section 4)—and reruns the value computation. The only information lost was the relation of the path before a cut edge with the path after the cut edge. Larger (64 bit) words would alleviate, though probably not eliminate, the need to truncate paths.

6 Experimental Results

This section uses the SPEC95 benchmarks to compare path profiling (PP) against edge profiling (QPT2), which has the lowest overhead of conventional profiling techniques [BL94]. The programs ran stand-alone on a Sun Ultraserver E5000–167Mhz UltraSPARC processors and 2GB of memory–running Solaris 2.5.1 with a local file system. Table 1 presents measurements of the SPEC95 benchmarks using the *ref* input data.[2] C benchmarks were compiled with gcc (version 2.7.1) and Fortran benchmarks were compiled with Sun's f77 (version 3.0.1). Both compilers used only the -O option.

PP's overhead (across all input files) averaged 30.9% (5.5–96.9%) and QPT2's overhead averaged 16.1% (-2.6–52.8%). PP's overhead averaged 2.8 times QPT2's overhead (0.7–14.5). PP's overhead is explained in part by the final two columns in Table 1, which report the fraction of path increments that required hashing and the average number of instructions between increments. However, the table does not report the cache interference caused by profiling code and data. In general, programs with little hashing (e.g., compress, li, ijpeg, turb3d) have PP overhead comparable or lower than QPT2. Programs with considerable hashing (e.g., tomcatv, fpppp, and wave5) can still have low overheads if blocks are large or paths are long and path increments execute infrequently.

Table 2 reports some characteristics of the program's acyclic paths. In all programs, the number of executed paths was small (fewer than 2300 in all except 099.go and 126.gcc) and is dwarfed by the potential paths–which number hundreds of millions to tens of billions, even after path truncation.

The table also reports the length of the longest and average acyclic paths. Not surprisingly, the weighted number of instructions in a path in the CFP95 benchmarks, 91.7 (43.1–636.0), is significantly longer than CINT95 benchmarks, 21.4 (15.1-33.2). More aggressive compiler optimizations would further increase these numbers by loop unrolling and procedure in-lining.

[2]Since PP measures a single process's execution, the tables report program behavior for each benchmark's last input file.

Benchmark	Base Time (sec)	PP Overhead %	QPT2 Overhead %	PP/ QPT	Path Inc (million)	Edge Inc (x Path)	Hashed Inc %	Inst/ Inc
099.go	885.0	53.4	24.1	2.2	1002.4	1.5	27.7	33.2
124.m88ksim	571.0	35.6	18.7	1.9	4824.9	1.2	3.9	16.2
126.gcc	322.0	96.9	52.8	1.8	9.4	1.7	16.8	15.1
129.compress	351.0	19.4	21.9	0.9	3015.7	1.5	0.0	16.6
130.li	480.0	25.4	26.7	1.0	3282.4	1.4	1.2	16.8
132.ijpeg	749.0	17.4	16.3	1.1	1164.9	1.1	1.2	31.0
134.perl	332.0	72.9	51.5	1.4	1133.0	1.9	23.4	22.2
147.vortex	684.0	37.7	34.1	1.1	3576.3	1.5	23.7	20.3
CINT95 Avg:		44.8	30.8	1.4	22251.1	1.5	12.2	21.4
101.tomcatv	503.0	19.9	2.8	7.1	574.6	1.1	95.8	93.0
102.swim	691.0	8.4	0.6	14.5	163.4	1.0	0.2	162.9
103.su2cor	465.0	10.1	5.8	1.7	558.1	1.2	21.5	92.8
104.hydro2d	811.0	37.7	5.8	6.5	1690.7	1.7	77.8	43.1
107.mgrid	872.0	6.3	3.2	2.0	1035.2	1.0	7.7	133.5
110.applu	715.0	71.0	12.0	5.9	2111.4	1.1	99.1	44.8
125.turb3d	1066.0	5.5	7.4	0.7	2952.8	1.1	0.0	56.5
141.apsi	492.0	7.7	1.8	4.2	599.3	1.1	3.5	84.0
145.fpppp	1927.0	14.6	-2.6	-5.6	395.0	1.8	42.5	636.0
146.wave5	620.0	16.9	6.1	2.8	737.3	1.3	65.0	74.1
CFP95 Avg:		19.8	4.3	4.0	1081.8	1.2	41.3	142.1
Average:		30.9	16.1	2.8	1601.5	1.3	28.4	88.4

Table 1. Comparison of path profiling (PP) against Ball-Larus edge profiling (QPT2). *Base Time* is elapsed time of uninstrumented program with the *ref* dataset. Overhead is the increase in execution time due to profiling. *PP/QPT2* is the ratio of the overheads. The remaining numbers report a program's behavior on its last input file. *Path Inc* is the number of increments of path counters. *Edge Inc* is the ratio of edge profiling increments to path increments. *Hash Inc* is the fraction of path increments that required hashing. *Inst/Inc* is the average number of instructions between increments.

Benchmark	Num	Path Profile				%	Edge Profile Paths				Routines		
		Longest		Avg			Longest		Avg		Exec	Max	Avg
	Path	Edge	Inst	Edge	Inst	Correct	Edge	Inst	Edge	Inst		Path	Path
099.go	24414	105	314	10.9	33.2	4.3	84	252	5.0	19.3	407	1574	60.0
124.m88ksim	1113	138	360	5.8	16.2	29.8	138	360	4.3	9.1	220	70	5.1
126.gcc	9319	711	1074	7.4	15.1	20.8	711	1074	4.6	10.5	1027	163	9.1
129.compress	249	80	146	6.5	16.7	43.0	80	146	4.6	8.8	69	34	3.6
130.li	770	153	252	9.0	16.8	38.1	62	109	7.0	14.6	216	64	3.6
132.ijpeg	1199	139	416	7.0	31.0	36.4	139	202	5.1	22.2	252	194	4.8
134.perl	1421	123	305	10.7	22.2	24.5	115	207	7.3	16.7	233	125	6.1
147.vortex	2223	584	841	8.9	20.3	39.5	584	841	8.5	13.6	627	65	3.5
CINT Avg:	5088	254	464	8.3	21.4	29.5	239	399	5.8	14.4	381	286	12.0
101.tomcatv	421	83	326	4.1	93.0	49.6	82	201	3.9	25.2	146	56	2.9
102.swim	378	106	310	2.3	162.9	57.1	106	310	2.2	57.6	143	25	2.6
103.su2cor	905	136	954	6.7	92.8	45.9	73	781	5.5	59.3	209	127	4.3
104.hydro2d	1456	344	488	6.5	43.1	33.2	344	436	5.8	36.2	227	434	6.4
107.mgrid	589	83	320	2.3	133.5	44.8	78	320	2.1	15.8	160	73	3.7
110.applu	619	240	3557	3.7	44.8	54.1	240	3557	2.4	26.6	144	82	4.3
125.turb3d	674	162	692	7.1	56.5	46.6	162	692	5.1	28.2	189	39	3.6
141.apsi	1064	712	1196	6.1	84.0	40.8	136	734	4.6	69.0	242	54	4.4
145.fpppp	821	85	11455	14.9	636.0	25.8	76	11455	9.0	122.6	143	322	5.7
146.wave5	896	90	1180	5.2	74.1	47.8	90	1180	4.8	49.5	212	69	4.2
CFP Avg:	782	204	2048	5.9	142.1	44.6	139	1967	4.5	49.0	182	128	4.2
Average:	2696	226	1344	6.9	88.4	37.9	183	1270	5.1	33.6	270	198	7.7

Table 2. Characteristics of executed acyclic paths. Paths are from the last input file in the *ref* dataset. *Num Paths* is the number of paths executed. *Longest* is the longest executed path. *Avg* is the average path length, weighted by execution frequency. *% Correct* is the fraction of paths predicted entirely correctly by edge profiling. Statistics on edge-predicted paths are up to the first misprediction. The final columns report the executed routines, and maximum and average number of paths per executed routine.

Benchmark	Common Paths			Common Instructions	
	Number	Static	Dynamic	Number	Dynamic
099.go	13670	52.8%	99.5%	32823357655	98.6%
124.m88ksim	800	71.1%	97.1%	68794935104	93.3%
126.gcc	9058	63.5%	94.6%	140747413	88.9%
129.compress	201	78.2%	99.2%	49206874856	98.0%
130.li	522	67.8%	87.4%	48285966561	87.7%
132.ijpeg	1099	85.3%	99.9%	36098785257	100.0%
134.perl	687	41.8%	71.9%	16041579783	63.8%
147.vortex	2160	96.0%	100.0%	72589443060	100.0%
101.tomcatv	418	90.9%	100.0%	53422202701	100.0%
102.swim	373	98.2%	100.0%	26607743709	100.0%
103.su2cor	880	95.5%	100.0%	51766673091	100.0%
104.hydro2d	1336	90.1%	100.0%	72892099182	100.0%
107.mgrid	577	97.3%	100.0%	138157253535	100.0%
110.applu	582	87.5%	100.0%	94622106074	100.0%
125.turb3d	651	94.5%	100.0%	166724577396	100.0%
141.apsi	992	91.0%	91.2%	37648860455	74.3%
145.fpppp	764	91.9%	99.9%	250229811735	99.6%
146.wave	5821	88.5%	98.8%	52996842799	96.9%

Table 3. Comparison of paths with the SPEC95 *ref* and *train* input datasets. *Common Paths* reports the number of paths executed for both input datasets. *Static* is the fraction of executed *ref* dataset paths executed by the *train* dataset. *Dynamic* is this fraction, weighted by execution frequency. *Common Instructions* reports the number and fraction of instructions along common paths.

The table also compares paths predicted using edge profiling against measured paths. For this experiment, PP followed each executed path, starting at a function entry or loop head, and used the most frequently executed edge out of a block to predict the next step in a path. If two edges had the same frequency, PP followed the not taken edge. This approach predicts, in their entirety, an average of 37.9% (4.3–57.1%) of the paths. PP also computed the length of a predicted path, up to their first mispredicted edge. Weighted by execution frequency, edge predicted paths were nearly as long as measured paths (5.1 versus 6.9 CFG edges), but contained significantly fewer (33.6 versus 88.4) instructions. In part, this result reflects the simple behavior of the SPEC95 benchmarks, which execute an average of 7.7 (2.6–60.0) paths in each routine. It is also consistent with previous work in dynamic and static branch prediction [FF92, BL93], which found that branches in a program typically follow one direction with very high probability, and this high-probability direction generally remains the same for different inputs.

Table 3 compares paths executed with the SPEC95 *train* and *ref* input datasets. In the CINT benchmarks, the *train* dataset identified 41.8–96.0% of the paths executed in the *ref* dataset. However, these common paths were the important ones that account for 71.9-100.0% of path executions and 63.8–100.0% of instruction executions. The CFP benchmarks were even more striking, as the *train* dataset identified a minimum of 87.5% of the executed paths, which in all except one case (147.apsi) account for 99+% of instructions. This result is again consistent with earlier work that found program behavior to be independent of program data [FF92, BL93]. These measurements suggest that path profiles could greatly improve peak SPEC95 performance by providing an inexpensive and accurate basis for profile-driven compilation.

7 Summary

This paper describes a new algorithm for path profiling. A path profile records the execution frequencies of acyclic paths in a routine. Although these profiles provide far more information than basic block or edge profiles, they can be obtained at a cost comparable to the best known profiling algorithms.

We also compared path profiles against paths predicted by a common heuristic that selects the highest frequency edge out of a basic block. Path profiles yielded slightly longer paths that contained significantly more instructions. If edge profiling was significantly cheaper or easier than path profiling, it might be a reasonable heuristic. However, since path profiling is similar in cost and complexity to edge profiling, there is little reason to forgo the benefits of measurement.

Moreover, path profiles obtained from short runs on short, training datasets covered most of the paths and instructions found in full runs. The profiles from these short runs form an inexpensive and accurate basis for profile-driven compilation.

Acknowledgements

Many thanks to Vasanth Bala for helpful discussions about path profiling. Chris Lukas and David Wood provided helpful comments on this paper.

References

[Bal94] Thomas Ball. Efficiently counting program events with support for on-line queries. *ACM Transactions on Programming Languages and Systems*, 16(5):1399–1410, September 1994.

[Bal96] Vasanth Bala. Low overhead path profiling. Technical report, Hewlett Packard Labs, 1996.

[BL93] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 300–313, June 1993.

[BL94] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[Cha88] Pohua P. Chang. Trace selection for compiling large C application programs to microcode. In *21th Annual Workshop on Microprogramming and Microarchitecture (MICRO 21)*, pages 21–29, November 1988.

[Ell85] John R. Ellis. Bulldog: A compiler for VLIW architectures. Technical Report YALEU/DCS/RR-364, Yale University, Department of Computer Science, February 1985.

[FF92] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 85–95, October 1992.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[Gol91] Aaron Goldberg. Reducing overhead in counter-based execution profiling. Technical Report CSL-TR-91-495, Computer System Laboratory, Stanford University, October 1991.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[YS94] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 232–241, October 1994.