

Selective Path Profiling

Taweessup Apiwattanapong
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
term@cc.gatech.edu

Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
harrold@cc.gatech.edu

ABSTRACT

Recording dynamic information for only a subset of program entities can reduce monitoring overhead and can facilitate efficient monitoring of deployed software. Program entities, such as statements, can be monitored using probes that track the execution of those entities. Monitoring more complicated entities, such as paths or definition-use associations, requires more sophisticated techniques that track not only the execution of the desired entities but also the execution of other entities with which they interact. This paper presents an approach for monitoring subsets of one such program entity—acyclic paths in procedures. Our *selective path profiling* algorithm computes values for probes that guarantee that the sum of the assigned value along each acyclic path (path sum) in the subset is unique; acyclic paths not in the subset may or may not have unique path sums. The paper also presents the results of studies that compare the number of probes required for subsets of various sizes with the number of probes required for profiling all paths, computed using Ball and Larus' path profiling algorithm. Our results indicate that the algorithm performs well on many procedures by requiring only a small percentage of probes for monitoring the subset.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—Monitors, Testing tools

General Terms

Algorithms, Experimentation

Keywords

Path profiling, dynamic analysis, static analysis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASTE'02, November 18–19, 2002, Charleston, SC, USA.
Copyright 2002 ACM 1-58113-479-7/02/0011 ...\$5.00.

1. INTRODUCTION

Dynamic program information, gathered during program execution, is important for many compiling and software engineering tasks. For example, profile-based optimization can improve program performance (e.g., [6]), coverage-based testing can help uncover faults in the software (e.g., [9]), and dynamically determined program invariants can help understand the behavior of the program (e.g., [8]). To utilize the power of these techniques fully, the dynamic information must represent the behavior of the program in the field.

Existing techniques for gathering dynamic information have several problems. First, the overhead in terms of time and space incurred by the probes can be unacceptable to users. Studies have shown that this instrumentation can increase the program's execution time by more than 100% [11]. Furthermore, the increase in space requirements for instrumented programs can be prohibitive for systems, such as embedded systems, that have rigid memory requirements. Second, providing accurate dynamic information about software is difficult to perform in-house. The many potential execution environments, lack of knowledge about users' operational profiles, and limited resources make gathering accurate information in-house a difficult, if not impossible, task.

Researchers have investigated ways to solve these problems using a number of techniques. To reduce the overhead, some techniques perform static analyses that reduce the number of probes required to gather the dynamic information (e.g., [2]). Other techniques gather the dynamic information using sampling techniques (e.g., [1]) that record a subset of the events that occur. Such techniques do reduce the overhead caused by instrumentation but may still be too expensive for monitoring the software in the field. Another technique, *residual testing*, captures coverage information after deployment about entities, such as statements and branches, that were not executed by the in-house testing [14]. As entities are executed after deployment, residual testing can remove appropriate probes. However, the technique cannot easily remove the probes required for more complicated coverage, such as path coverage, to provide monitoring of only the remaining uncovered entities. Finally, techniques, such as our GAMMA technology [5, 13], that provide continuous monitoring of software after deployment are being developed. These techniques use lightweight instrumentation that assigns different monitoring tasks to different instances of the deployed software. The information gathered from different instances is then synthesized to give the overall desired monitoring information. However, to use this technology, techniques that can monitor a subset of the program entities in the deployed software are required.

The need for such lightweight instrumentation techniques motivates a new way of monitoring that insert probes to gather information about a subset of the entities in the program. For some tasks, such as branch profiling, the method is simple—insert a probe that will record the execution of that entity. For more complicated monitoring, such as path profiling or data-flow coverage, this simple approach will not work. For example, to record information about the acyclic paths in a procedure that have executed, information about the execution of other paths must be known. Similarly, for measuring coverage of definition-use associations, probes must record not only the execution of the desired definitions and uses but also the execution of any potential intervening definitions.

In this paper, we present such a selective approach for a powerful profiling technique—acyclic path profiling. Our new approach, *selective path profiling*, works on a subset of acyclic paths in a procedure, and computes the edge probes required to monitor the subset. We present the algorithm in detail and illustrate it with an example. To evaluate our technique, we implemented our algorithm and performed a set of studies. For comparison, we also implemented Ball and Larus’ efficient path profiling algorithm [3]. Our studies show that, for our subject program, our algorithm can provide significant savings when we are interested in a subset of the acyclic paths in a procedure.

The main benefit of our algorithm is that it facilitates lightweight monitoring of an important type of program entity—acyclic paths. With our algorithm, the monitoring overhead, in terms of probes, consists of only those probes that are required to track the paths of interest. Because some large programs may have more than 2^{32} acyclic paths [3], profiling only those of interest can provide significant savings. Another benefit of our approach is that the acyclic paths can be used to provide additional dynamic information. For example, acyclic paths can be used to identify those definition-use associations that are executed efficiently and only the subset of acyclic paths required to track the definition-use pairs of interest need be monitored. A third benefit of our approach is that it provides a way to perform residual monitoring for path coverage effectively. Our algorithm can be used to determine the required probes for the sets of uncovered paths. Finally, the algorithm facilitates the GAMMA technology by identifying probes required for subsets of acyclic paths in the program. Probes for these subsets can be assigned to different instances to provide distributed, lightweight monitoring of deployed software.

2. EFFICIENT PATH PROFILING

The efficient path-profiling algorithm, developed by Ball and Larus (BL), instruments a program so that, as the program executes, it records the number of times each acyclic path is executed [3]. The algorithm consists of four steps, which are performed on each procedure that is to be monitored: (1) creation of the representation, (2) assignment of values to edges in the representation, (3) selection of edges to be instrumented and computation of increments for instrumented edges, and (4) optimization of the instrumentation. This section discusses Steps 1 and 2 in detail: Step 1 because our algorithm uses the representation created in this step; Step 2 to contrast the full instrumentation technique with our algorithm.

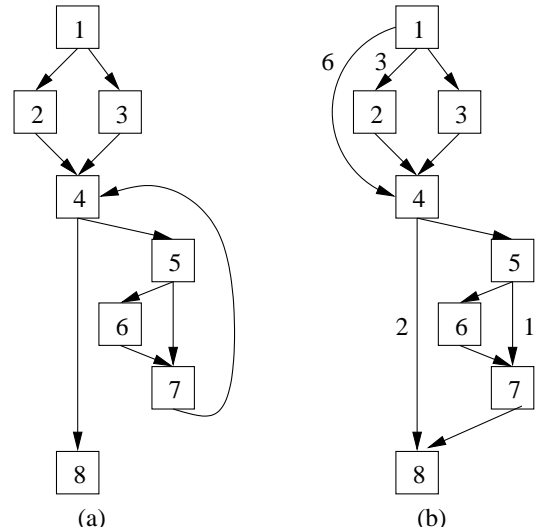


Figure 1: (a) Control-flow graph and (b) derived DAG with edge values computed using BL algorithm.

In the first step, BL transforms the control-flow graph (CFG)¹ of the procedure to a directed acyclic graph (DAG). To do this, for each backedge in the CFG, the algorithm performs three actions:

- It adds a dummy edge from the entry node (*ENTRY*) to the target of the backedge.
- It adds a dummy edge from the source of the backedge to the exit node (*EXIT*).
- It removes the backedge.

To illustrate, consider the CFG in Figure 1(a) and its associated DAG in Figure 1(b). The CFG has one backedge (7,4). In the DAG, this backedge has been replaced by edges (1,4) and (7,8). Note that nodes 1 and 8 are the *ENTRY* and *EXIT* nodes, respectively.

In the second step, the algorithm assigns values to the edges in the DAG in such a way that the sum of values along each path is unique. To do this, the algorithm visits nodes in the DAG in reverse topological order. A value $NumPaths(v)$, which represents the number of paths from v to *EXIT*, is associated with each node v . The algorithm visits all n outgoing edges from v , and assigns a value, $Val()$, to the k^{th} outgoing edge (v, w_k) , where $Val(v, w_k) = \sum_{i=1}^{k-1} NumPaths(w_i)$.

To illustrate, consider node 5 in Figure 1(b). Before visiting this node, the algorithm visits nodes 8, 7, and 6, in that order, and computes $NumPaths(v)$ for each node. At this point in the computation, $NumPaths(8)$ is 1, $NumPaths(7)$ is 1, and $NumPaths(6)$ is 1. The algorithm then visits one of the outgoing edges from node 5; assume that this outgoing edge is (5,6). Because edge (5,6) is the first outgoing edge visited, applying the formula given above to compute $Val(5,6)$, where $k=1$ and w_1 is node 6, yields $Val(5,6) = 0$. Next, the algorithm visits edge (5,7). Because (5,7) is

¹A *control-flow graph* is a directed graph in which nodes represent statements and edges (possibly labeled) represent flow of control between statements. CFG has a unique entry node that has no predecessors and a unique exit node that has no successors.

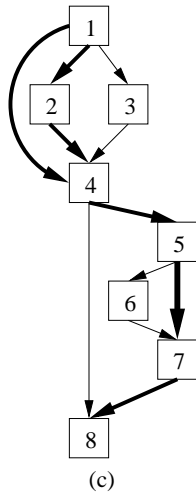


Figure 2: DAG of Figure 1(b) with two paths of interest highlighted in bold.

the second edge (i.e., $k=2$), $Val(5,7) = NumPaths(6) = 1$. $NumPaths(5)$ is the sum of the $NumPaths(s)$, for all s that are successors of 5. Thus, $NumPaths(5)$ is 2. The algorithm then continues by visiting the rest of the nodes in the DAG. The resulting $Val()$ for edges in the DAG are shown as labeled edges in the DAG in Figure 1(b); unmarked edges, such as (1,3) and (4,5), have $Val() = 0$.

3. SELECTIVE PATH PROFILING

The goal of selective path profiling is to monitor the execution of a subset of the acyclic paths in the program. We call the paths in this subset the *paths of interest* and refer to it as *PI*. We call those paths not in this subset the *paths not of interest* and refer to it as *PN*. Clearly, *PI* and *PN* form a partition of the set of paths in the DAG. To illustrate, consider Figure 2. Suppose that there are two paths of interest (i.e., in *PI*) in the DAG: path 1,2,4,5,7,8 and path 1,4,5,7,8. The edges in these paths are shown in bold in the figure. There are seven other paths, such as path 1,2,4,8 and path 1,3,4,5,7,8, which, in this case, are paths not of interest (i.e., in *PN*).

Before developing our algorithm, we investigated the use of several other approaches, such as complete edge profiles and complete path profiles, for profiling acyclic paths selectively. However, none of these approaches is effective or efficient for producing accurate profile of some selected paths. We also considered using Ball and Larus' algorithm applied only to the paths in *PI*. To do this, we created a subgraph of the DAG by removing those edges in the DAG that are not in *PI*. The resulting subgraph for the DAG in Figure 1(b) contains edges (1,2), (2,4), (4,5), (5,7), (7,8), and (1,4) is shown in Figure 3.

If we apply Ball and Larus' algorithm to this subgraph, it will process the nodes in the order 8, 7, 5, 4, 2, and 1. Because there is only one path in *PI* from node 2 to *EXIT*, $NumPaths(2)$ will be 1 and edges (2,4), (4,5), (5,7), and (7,8) will have $Val()$ of 0. When the algorithm processes node 1, it first processes edge (1,4) and assigns 0 to $Val(1,4)$. When the algorithm processes the other edge, (1,2), it assigns 1 to $Val(1,2)$. With these edge assignments,

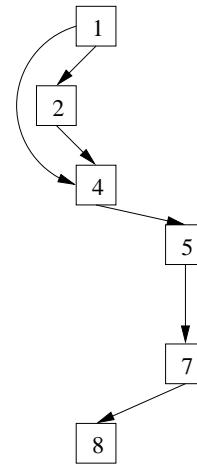


Figure 3: Subgraph of the control-flow graph given in Figure 1(b) that consists only of those paths of interest.

the sum obtained for path 1,4,5,7,8 is 0 and the sum for path 1,2,4,5,7,8 is 1. These sums, however, are not unique when considered in the context of the original DAG in which $Val()$ of edges not in the subgraph are 0. For example, paths 1,3,4,8 and 1,3,4,5,6,7,8 also have sum 0. If the path sum of a path is not unique, the profile of that path cannot be determined. Thus, this modification does not provide a method for recording the execution of only the paths in *PI*.

The Algorithm

To compute unique path sums for only paths of interest efficiently, we adapted Ball and Larus' algorithm by adding a constraint on the order of visiting the outgoing edges from a node; we then process the nodes in the DAG using their algorithm. Like Ball and Larus' algorithm, a probe is assigned to the edge whose $Val() \neq 0$. After probes have been initially assigned to the edges, we push probes down to other edges in the graph. Because some probes can be combined with others or can be removed in the next phase, we call this the optimization of the placements of probes. Finally, we remove the probes from those edges that are not in any paths in *PI*. The result is a unique path sum for each path in *PI*; the path sums for paths in *PN* may not be unique.

Figure 4 presents our selective path profiling algorithm (SPP). Like Ball and Larus' algorithm, our algorithm uses the DAG representation for the CFG and computes the edge values $Val()$ for edges in the DAG. The algorithm also takes the set of edges in all paths in *PI* as another input; we call this set *EI*.

In the first phase of SPP (lines 1-12), the algorithm initializes the number of paths ($NumPaths(EXIT)$) at *EXIT* node to 1. The algorithm then visits each non-exit node v in reverse topological order. At each node v , SPP initializes $NumPaths(v)$ to 0. It then visits each outgoing edge e not in *EI*, assigns $Val(e)$ and updates the number of paths from v to *EXIT* ($NumPaths(v)$) to include the number of paths starting from that edge. Likewise, SPP visits each edge in *EI* and performs the same operations. The algorithm continues processing the next nodes until all nodes are visited. By

```

Procedure SPP( $G, EI$ )
input     $G$ : DAG to be profiled
         $EI$ : set of edges in all paths in  $PI$ 
output   $Val(e)$ : edge value for each edge  $e$  in  $G$ 
declare  $NumPaths(v)$ : number of paths
        from node  $v$  to  $EXIT$ 
begin SPP
    /* phase 1: compute edge values */
1.   $NumPaths(EXIT) = 1$ 
2.  foreach non-exit node  $v$  in reverse topological order do
3.     $NumPaths(v) = 0$ 
4.    foreach outedge  $e$  not in  $EI$ :  $v \rightarrow w$  do
5.       $Val(e) = NumPaths(v)$ 
6.       $NumPaths(v) = NumPaths(v) + NumPaths(w)$ 
7.    endfor
8.    foreach outedge  $e$  in  $EI$ :  $v \rightarrow w$  do
9.       $Val(e) = NumPaths(v)$ 
10.      $NumPaths(v) = NumPaths(v) + NumPaths(w)$ 
11.   endfor
12. endfor
    /* phase 2: optimize the placement of probes */
13. foreach node  $v$  in topological order do
14.   if ( $v$  has only one inedge  $ei$ ) and ( $Val(ei) > 0$ ) then
15.     foreach outedge  $eo$ :  $v \rightarrow w$  do
16.        $Val(eo) = Val(eo) + Val(ei)$ 
17.     endfor
18.      $Val(ei) = 0$ 
19.   endif
20. endfor
    /* phase 3: remove probes on edges not in  $EI$  */
21. foreach edge  $e$  not in  $EI$  do
22.    $Val(e) = 0$ 
23. endfor
end SPP

```

Figure 4: Selective path profiling algorithm.

visiting the edges not in EI first, the algorithm guarantees that $Val(e)$ of edges in EI are greater than those of edges not in EI (if any). By keeping the path sums of paths in PI higher than those not in PI , the algorithm guarantees that the path sums of the paths in PI are unique. When probes on the edges not in EI are removed, the path sums of the PN paths containing those edges are reduced and may be the same as the path sums of other PN paths. However, the path sums of the PI paths are unaffected and still unique. Using this basic idea, every branch node keeps the path sums of paths in PI starting from that node unique so that, when the $ENTRY$ node is reached, all paths in PI have unique path sums.

In the next phase of SPP (lines 13-20), the algorithm optimizes the placement of the probes. To do this, the algorithm visits each node in the DAG in topological order. For each node that has only one incoming edge (ei) whose $Val(ei)$ is greater than zero, the algorithm increments $Val(eo)$ of all outgoing edges (eo) by $Val(ei)$. Then, the algorithm sets $Val(ei)$ to zero.

In the last phase of SPP (lines 21-23), the algorithm removes the probes on edges not in EI . Performing this step is straightforward. The algorithm visits each edge e not in EI , and resets $Val(e)$ to zero.

The result is a set of unique path sums for paths in PI .

Example

To illustrate how SPP works, consider the example in Figure 5(a). Paths 1,2,4,5,7,8 and 1,4,5,7,8 are paths of interest and are highlighted in bold. In this case, EI is $\{(1,2), (1,4), (2,4), (4,5), (5,7), (7,8)\}$.

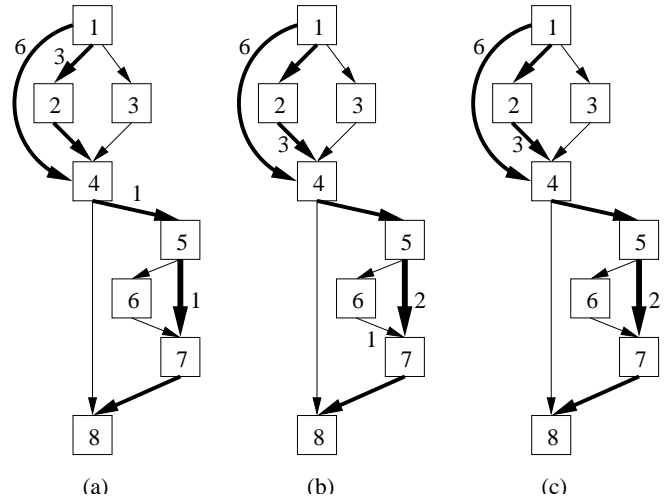


Figure 5: DAG of Figure 1(b) with two paths of interested in bold after processing each step: (a) compute edge values using algorithm SPP (b) optimize the placement of probes and (c) remove probes on edges not in EI

In the first phase of the algorithm, SPP processes the nodes in a reverse topological order; assume that this order is 8,7,6,5,4,2,3,1. Because node 8 is the exit node and is already processed, resulting in $NumPaths(EXIT)$ being 1, the algorithm moves to node 7. $NumPaths(7)$ is initialized to 0. There is no outgoing edge that is not in EI at node 7. Thus, SPP visits the only outgoing edge, (7,8), and assigns 0 to $Val(7,8)$. $NumPaths(7)$ is updated to 1. The algorithm then visits node 6, initializes $NumPaths(6)$ to 0, and processes edge (6,7), which is not in EI . $Val(6,7)$ is set to 0 because $NumPaths(6)$ is 0, and $NumPaths(6)$ is set to 1. When SPP visits node 5, it initializes $NumPaths(5)$ to 0 and processes edge (5,6) first because that edge is not in EI . $Val(5,6)$ is assigned 0, and $NumPaths(5)$ is now 1. The algorithm processes the other edge (5,7), sets $Val(5,7)$ to 1 and updates $NumPaths(5)$ to 2. Similarly, SPP visits node 4, initializes $NumPaths(4)$ to 0, visits edge (4,8), sets $Val(4,8)$ to 0, and updates $NumPaths(4)$ to 1. It then processes edge (4,5), sets $Val(4,5)$ to 1 and $NumPaths(4)$ to 2. SPP visits node 2, 3, and 1, respectively, and computes $Val()$ and $NumPaths()$ for those edges and nodes. Note that at node 1, there are two outgoing edges that appear in EI . SPP can visit either of them first. The edge values shown in Figure 5(a) are computed by assuming that edge (1,2) is visited before edge (1,4). The result of this step is the DAG with $Val()$ shown in Figure 5(a).

In the second phase, SPP visits the nodes in the graph to optimize the placement of the probes. In the example in Figure 5(a), $Val(1,2)$ is moved to edge (2,4) but cannot be moved past node 4 because node 4 has three incoming edges; in this case the new placement does not reduce the number of probes. The algorithm pushes $Val(4,5)$ to the edges leaving node 5, edges (5,6) and (5,7), and adds $Val(4,5)$ to $Val(5,6)$ and $Val(5,7)$. This move reduces the number of probes because the probe on edge (5,6) is pushed further to edge (6,7) and will be removed in the last phase. The result is shown in Figure 5(b).

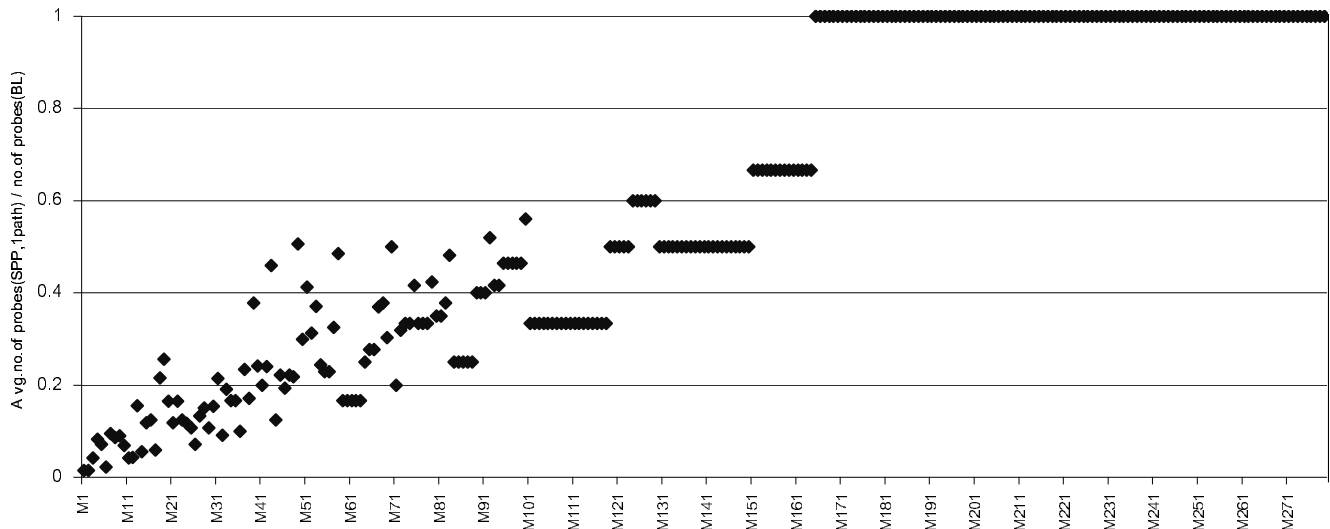


Figure 6: The average number of probes for one path computed by SPP per the number of probes for all paths computed by BL for each method in `javac`. Methods on the horizontal axis are sorted by the ratio of the number of paths in the method to $2^{|\text{probes}(BL)|}$

Finally, in the last phase, SPP visits the edges in the graph and removes probes on edges not in EI . For the example in Figure 5, the probe on (6,7) is removed. The result is shown in Figure 5(c).

After completion of the processing, Path 1,2,3,4,5,7,8 and path 1,4,5,7,8 have the path sums of 5 and 8, respectively.

Worst-Case Complexity

The worst-case time complexity of SPP depends on the time required for each of the three phases of the algorithm: (1) computing edge values using SPP (2) optimizing the placement of probes, and (3) removing probes on edges not in EI . The first phase requires a reverse topological sort. It also requires that each edge e in G be visited once. The second phase requires that each edge e in G be visited twice. The third phase requires that each edge e in G be visited once. Thus, the worst-case time complexity of SPP is $O(e)$ where e is the number of edges in G .

The worst-case space complexity of SPP depends on the space needed for the DAG representation, the number of paths ($NumPaths(v)$) for each node v , the set of edges EI , and the edge values $Val(e)$. The number of nodes in a CFG is bounded by the number of edges + 1; this also holds for the DAG representation of a CFG. Thus, the space required by $NumPaths(v)$ is bounded by $e + 1$. The space required by EI , $Val(e)$, and the DAG representation is linear in e . Thus, the worst-case space complexity of SPP is $O(e)$, where e is the number of edges in G .

4. EMPIRICAL STUDIES

To evaluate our approach, we implemented SPP and conducted two studies. For comparison, we also implemented Ball and Larus' algorithm (BL). In this section, we describe the subject that we used, and for each study, present the goals and methods along with the results and conclusions of the study.

For both studies, we used the number of probes needed for profiling one or more paths in the DAG of a method as an

indicator of the overhead that the instrumentation incurs. We measured the number of probes required by our algorithm for profiling a path or a set of paths and the number of probes required by BL. We obtained the number of probes by counting the edges whose edge values are non zero. In the future, we plan to measure the actual overhead by executing the instrumented subjects.

The subject we used in both studies is `javac`, a java compiler developed at Sun Microsystems. We used a subset of `javac`; our subset consists of 465 methods in 20 classes. The numbers of acyclic paths in these methods range from 1-5656.

Study 1: One path

The goal of Study 1 was to measure the number of probes required by SPP for single paths in the DAG, compare it with the number of probes required by BL, and determine if savings in overhead could be obtained using SPP. To do this, for each method, we ran our implementation of SPP with each path in the DAG as input. For each path, we counted the number of probes required for profiling that path. After processing all paths, we computed the average number of probes and the ratio of this average to the number of probes computed by BL.

The graph in Figure 6 shows the results of this study. The horizontal axis represents 279 of the 465 methods in `javac`. We omitted the other 186 methods because each of them has only one path and thus, both SPP and BL require no probes. Note that, although there are points on the graph for all 279 methods, the horizontal axis is labeled in intervals of ten.

We sorted the 279 methods in increasing order by the ratio of the number of paths in the method to $2^{|\text{probes}(BL)|}$ where $|\text{probes}(BL)|$ is the number of probes required by BL. This ratio is related to the structure of the method: if this structure is a chain of `if-then-else` statements, the number of probes needed by BL is n and the number of paths in that method is 2^n . From our observation using manual inspections, the ratio represents how close the structure of

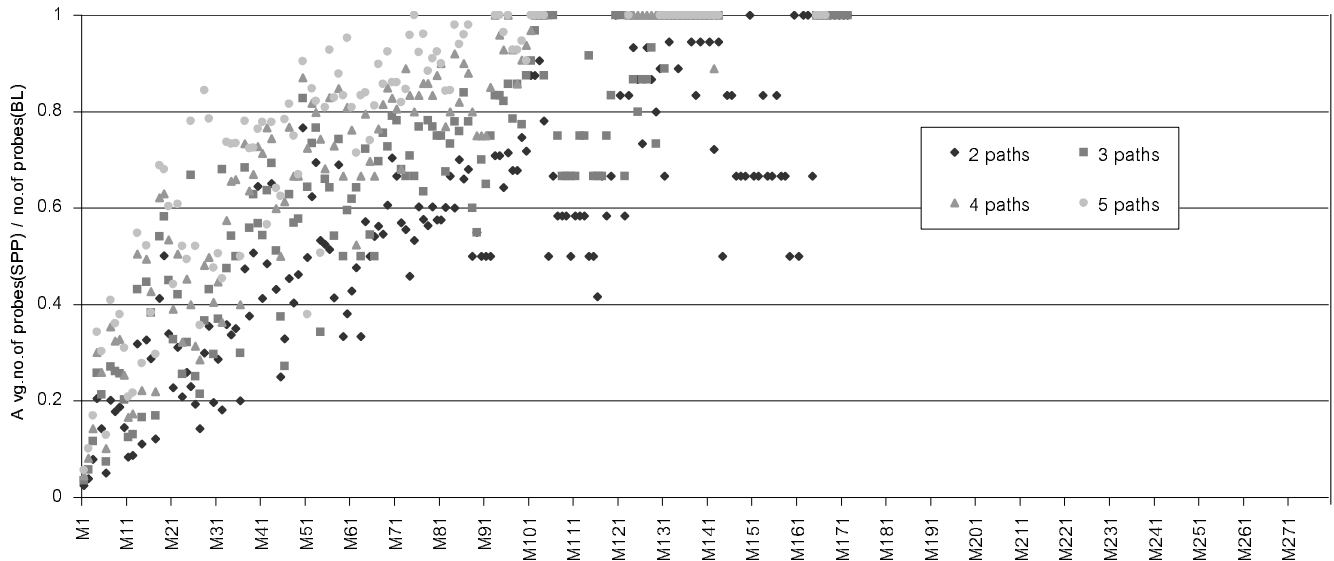


Figure 7: The average number of probes for 2-5 paths computed by SPP per the number of probes for all paths computed by BL for each method in javac. Methods on the horizontal axis are sorted by the ratio of the number of paths in the method to $2^{|probes(BL)|}$

a method is to the chain of `if-then-else` statements: if the ratio is 1, the structure is such a chain; as the ratio decreases, the structure is more tree-like.

The vertical axis in the graph shows the ratio of the average number of probes computed by SPP for one path and the number of probes by BL for all paths for each method under consideration.

From the graph, we can see that the structure of a method has a significant effect on the effectiveness of our algorithm. If the structure of a method is more tree-like, indicated by a lesser ratio of the number of paths in the method to $2^{|probes(BL)|}$, profiling only one path using our algorithm can significantly reduce the number of probes required. For example, for approximately half of the methods studied, the percentage of probes needed to profile one path is less than 60% and, for many of these methods, the percentage is significantly less.

On the other hand, if the structure is more like a chain, indicated by a ratio close to one, our algorithm may not reduce the number of probes. However, the space required to store the path profile is reduced. In addition, 93% of methods that are represented by points on the right of the graph (methods M165-M279) have two paths each. This means that there are only a few large methods, which have the chain structure, so we can expect that our algorithm will often reduce the number of probes for large methods.

Study 2: A small subset of paths

The goal of Study 2 was to measure the number of probes required by SPP for a subset containing a small number of paths in the DAG, compare it with the number of probes required by BL, and determine if savings in overhead could be obtained using SPP. To do this, we conducted a study similar to Study 1 except that we considered subsets of paths in the DAG. We randomly selected 100 or fewer² different

²If the number of paths is less than 11, there are less than 100 different subsets containing two paths.

subsets containing two distinct paths. For each method, we ran our implementation of SPP with each path subset in the DAG as input. For each subset, we counted the number of probes required for profiling that subset. After processing all subsets, we computed the average number of probes and the ratio of this average to the number of probes computed by BL. We repeated this process for subsets containing 3, 4, and 5 paths.

The graph in Figure 7 shows the results of this study. The points on the graph that represent the various-sized path subsets have different symbols and vary in brightness. The axes in the graph are similar to those in the graph for Study 1. Because most of the methods on the right of the graph (methods M173-M279) have two paths each, there is no subset containing exactly two paths that differs from all paths. Thus, these methods are omitted from this study. This is also the case for methods with 3, 4, and 5 paths when we studied groups of paths of those sizes.

Because SPP uses a set of edges as input, if a group of five paths covers most edges in the DAG, the number of probes needed to profile this group is nearly the same as the number of probes required by BL. The resulting graph shows that, for about 70% of methods in our subject, five randomly selected paths cover most of the edges in the DAG. In Section 6, we will discuss a method for selecting a subset of paths that should be profiled together.

5. RELATED WORK

Section 1 presented related work that attempts to reduce the overhead of monitoring using techniques such as reducing the number of required probes, removing probes after the entities they monitor have been executed, or sampling to record a subset of the events that occur. Other related work uses various techniques to predict which paths in the program are hot paths.

Duesterwald and Bala [7] present the NET scheme, which combines basic block profiling for only the path heads (basic

blocks that are the targets of back edges). They show that, with speculation, in most cases their scheme can predict hot paths very well. They also showed that their approach incurs substantially less overhead than path profiling in an online setting. Ball, Magata, and Sagiv [4] present algorithms to determine when the edge profile is a good predictor for hot paths in a program. Their algorithms can derive definite and potential hot paths from an edge profile. The results show that the set of potential hot paths given by the edge profiles is very close to the set of actual hot paths measured by path profiles for most executions of programs in the SPEC95 benchmarks. However, some other edge profiles may not be good predictors.

Although these techniques can be useful for predicting hot paths in a program, they cannot replace path profiling for path coverage. Some edge profiles can assure that some paths are executed (definite hot paths [4]). There are other paths or even all paths in some edge profiles whose coverage cannot be determined.

6. CONCLUSION AND FUTURE WORK

We have presented a new approach for selective monitoring that computes profiles for a subset of a procedure's acyclic paths. We implemented both our algorithm and Ball and Larus' algorithm, which provides efficient profiling of all acyclic paths in a procedure. Our studies compare the numbers of probes required for one path or a subset containing a small number of paths to the number of probes required for profiling all paths. The results of our studies show that, for our subject program, we can get significant savings using our algorithm when profiling a small subset of paths of interest; the results also show that the reduction is related to the structure of the procedure and thus, in some cases, our approach provides little savings.

Selective profiling enables software developers to collect the required profile information while incurring only the necessary cost for the probes. The approach provides an opportunity to tune the tradeoff between the additional profile information and the extra runtime overhead and disk space. The approach also provides a way to profile systems that have limited memory resources, limited disk space, or strict time constraints, and thus, cannot be fully instrumented. In complex profiling tasks that involved the interaction between entities, such as path profiling, the selective approach also profiles other entities not of interest; we call these *by-product* entities.

There are a number of interesting issues in optimizing selective path profiling that can be addressed. For example, we may be able to identify a group of paths that can be profiled together efficiently. To use SPP as a part of the GAMMA project, the paths in a procedure are divided into groups, and each group is profiled at a different site. From Study 2, we discovered that profiling two disjoint paths incurs more overhead than profiling two paths that share the same subpaths. Therefore, some heuristics are needed to identify these groups of paths so that the profiling task is load balanced.

In addition to optimization, we also plan to consider extending our algorithm in several ways. Like BL algorithm, SPP is an intraprocedural path profiling algorithm. We plan to extend it to profile interprocedural paths. Because even a small program can have a very large number of interprocedural paths, we expect our algorithm to reduce more overhead

when profiling them selectively. Currently, we have considered two different techniques based on BL algorithm. Melski and Reps proposed an interprocedural path profiling technique [12]. This technique builds an interprocedural DAG from the interprocedural CFG so that paths cross the procedure boundary but still end at loop backedges. An edge value is a linear function of the number of path at exit of a procedure, instead of a number. The advantage of this technique is that it is not necessary to build an inlined interprocedural CFG but the need for additional computation at each instrumented edge brings significant overhead cost.

Another technique, *whole program paths* (WPP), proposed by Larus [10], overcomes both procedure and loop backedge limitations by recording the interprocedural path numbers in sequence. The sequence of path numbers with additional codes for procedure entry and exit forms a trace of that execution. WPP keeps the simplicity of computing edge values but the trace data is large and data compression is needed. In addition to these problems, there are many open issues when combining these techniques with SPP such as how to handle two different paths of interest that are in different calling contexts, and what we can derive from a trace that has non-unique path numbers in the middle of an execution. We are investigating new techniques, based on SPP, to address these problems.

Acknowledgments

This work was supported in part by National Science Foundation awards CCR-9988294, CCR-0096321, EIA-0196145, and SBE-0123532 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission. The anonymous reviewers provided many comments that helped us improve the paper. Saurabh Sinha made several suggestions for improving the paper.

7. REFERENCES

- [1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 168–179, June 2001.
- [2] T. Ball. Efficiently counting program events with support for on-line queries. *ACM Transactions on Programming Languages and Systems*, 16(5):1399–1410, September 1994.
- [3] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, December 1996.
- [4] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 134–148, January 1998.
- [5] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the ACM SIGSOFT/SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, November 2002, (to appear).
- [6] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 95–105, May 2002.

- [7] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, November 2000.
- [8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.
- [9] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, May 2002.
- [10] J. R. Larus. Whole program paths. In *SIGPLAN Conference on Programming Language Design and Implementation*, volume 34, pages 259–269, May 1999.
- [11] D. Melski. *Interprocedural path profiling and the interprocedural express-lane transformation*. PhD thesis, University of Wisconsin, 2002.
- [12] D. Melski and T. W. Reps. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 47–62, March 1999.
- [13] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 65–69, July 2002.
- [14] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering*, pages 277–284, May 1999.