Quantifying the Effectiveness of Testing via Efficient Residual Path Profiling

Trishul M. Chilimbi* trishulc@microsoft.com
Aditya V. Nori† adityan@microsoft.com
Kapil Vaswani‡ kapil@csa.iisc.ernet.in

*Microsoft Research Redmond †Microsoft Research India ‡IISc Bangalore

May 2007
Technical Report
MSR-TR-2007-62

Writing correct programs is hard. Proving that they are correct is even harder. Consequently, testing is extensively used for uncovering bugs in large, complex software. Since testing software exhaustively is infeasible, well designed regression test suites aim to anticipate all reasonable software usage scenarios and generate test cases that exercise those behaviors. Unfortunately, testers today have no way of knowing how much of real-world software usage was untested by their regression suite. While collecting path profiles of deployed software would provide this information, profiling overheads preclude this. This often results in released software shipping with bugs that could have been detected with a better test suite. Recent advances in low-overhead path profiling provide the opportunity to rectify this deficiency and perform residual path profiling on deployed software. Residual path profiling identifies all paths executed by deployed software that were untested during software development. We extend prior research to perform low-overhead interprocedural path profiling. We demonstrate experimentally that low-overhead path profiling, both intraprocedural and interprocedural, provides valuable quantitative information on testing effectiveness. We also show that residual edge profiling is inadequate as a significant number of untested paths include no new untested edges.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

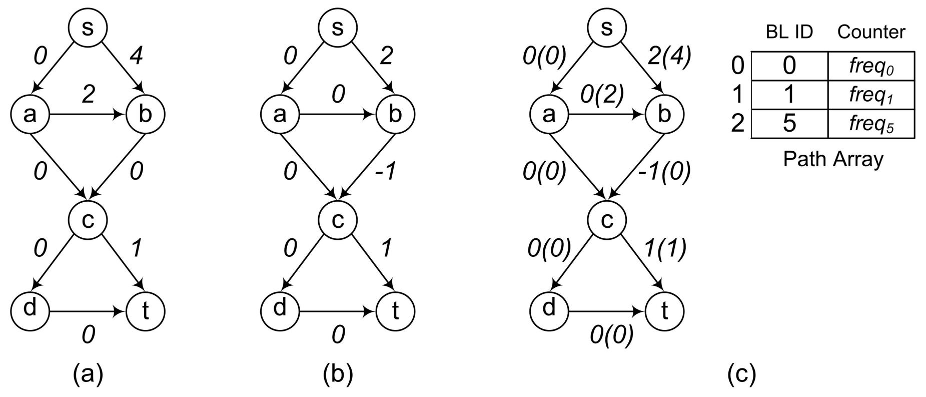
We are surrounded by software that runs on several different devices and performs a variety of tasks. Our growing reliance on software has increased the need to make software dependable. Unfortunately, writing correct programs is hard. Modern software development, which often involves a large team of potentially geographically separated programmers with varying abilities, compounds this problem.

Static analysis tools attempt to address this problem by checking that a program satisfies a set of specifications on all possible execution paths. As part of the verification process these tools produce execution traces that contain errors, if any exist. However, despite much recent progress in static verification, these have several limitations that preclude exclusive reliance on such techniques for large, complex software.

Consequently, static program checking is invariably complemented with testing, which runs a program on a suite of test cases and checks for errors. Unlike static checking tools that explore all program paths, testing can only detect errors along the set of paths that were executed. Since exhaustively testing large software is infeasible, well designed regression test suites aim to anticipate all reasonable software usage scenarios and generate test cases/inputs that exercise those behaviors. However, anticipating software usage is extremely hard especially when the same piece of software can run on a variety of devices. Ideally, one would like to profile actual software usage, perhaps during beta software testing, to detect untested software behaviors.

Program paths are a succint and pragmatic abstraction of a program's dynamic control flow behavior. They capture much more control flow information than basic block or edge profiles and are much smaller than complete instruction traces. Consequently, We would like to detect program paths executed during actual software usage that were untested, and use this information to improve the test suite, and consequently, testing effectiveness.

Unfortunately, collecting path profiles using the standard Ball-Larus technique incurs 50% overhead on average and upto 200% in the worst case [7]. Researchers have proposed a variety of techniques to lower this overhead further, but these techniques are only effective when the number of paths that must be profiled is small, as in the case of hot path profiles for optimization [4, 7]. Recently, researchers have proposed a technique called preferential path profiling that reduces the overhead of path profiling even when the number of profiled paths is large [17]. We show that preferential path profiling can be used to perform low-overhead residual path profiling, which identifies all paths executed by deployed software that were untested during software development [14]. Since preferential path profiling captures intraprocedural program paths, it is well suited for unit testing, which validates that individual modules are working properly by writing test cases for each function. However, it cannot be used for performing residual profiling of integration testing [3], which combines individual modules and tests them as a group, as this requires profiling interprocedural paths.



	BL ID	Counter
0	0	*freq*$_0$
1	1	*freq*$_1$
2	5	*freq*$_5$

Figure 1: Motivating example for PPP. (a) A DAG G with 6 paths with edges numbered using the Ball-Larus algorithm. (b) G with edges having only PPP assigned numbers for three interesting paths $I = \{sacdt, sact, sbct\}$. (c) G with edges assigned two numbers, (PPP$-number$, BL$-number$). The path array is accessed using the PPP number.

To address this limitation, we have extended preferential path profiling to perform low-overhead interprocedural path profiling. We have evaluated our residual path profiling scheme on several SPEC 2000 benchmarks, using the train inputs to model a test suite and the ref inputs to model field inputs. The results indicate that residual interprocedural path profiling incurs low overhead and detects a large number of untested paths. In addition, we show that residual edge profiling is inadequate as a significant number of untested paths include no new untested edges

The three main contributions of this paper are:

- Design and implementation of a low-overhead interprocedural path profiling scheme based on preferential path profiling (Section 3).
- Implementation of low-overhead residual path profiling for both intraprocedural and interprocedural program paths (Section 4).
- Evaluation results that demonstrate that residual path profiling provides valuable quantitative information on testing effectiveness. In addition, we show that residual edge profiling is inadequate as a significant number of untested paths include no new untested edges (Section 5).

2 Background: Preferential Path Profiling (PPP)

This section provides a brief overview of preferential path profiling [17]. The Ball-Larus profiling scheme, which forms the basis of many path profilers, assigns weights to edges of a control flow graph (CFG) such that all paths are allocated unique identifiers (i.e., the sum of the weights of the edges along every path is unique) [2]. During program execution, the profiler accumulates weights along the edges and updates an array entry that corresponds to this path identifier. Unfortunately, for functions with a large number of paths, allocating an array for all program paths is prohibitively expensive, if not infeasible. Consequently, path profiler implementations are forced to use a hash table to record path information for such functions. Although using a hash table is space efficient as programs typically execute only a small subset of all possible paths, it incurs significantly higher execution time overhead as compared to updating an array entry. Previous work has shown that hash tables account for a significant fraction of the overhead attributable to path profiling [7].

Preferential path profiling (PPP), efficiently profiles arbitrary path subsets, which are referred to as interesting paths. As mentioned earlier, the Ball-Larus algorithm assigns weights to the edges of a given CFG such that the sum of the weights of the edges along each path through the CFG is unique. PPP generalizes this notion to a subset of paths; it assigns weights to the edges such that the sum of the weights along the edges of the interesting paths is unique. Furthermore, PPP attempts to achieve a minimal and compact encoding of the interesting paths; such an encoding significantly reduces the overheads of path profiling by eliminating expensive hash operations during profiling. In addition, PPP separates interesting paths from other paths, and this enables residual path profiling as we show in Section 4.

We use an example from [17], shown in Figure 1 to illustrate how PPP works. The DAG G in Figure 1(a) is obtained from the CFG of the function. The figure shows the weights assigned by the Ball-Larus algorithm to the edges of G. Note that the sum of the weights of edges along every path from the start node s to the final node t is unique, and all paths are allocated identifiers from 0 to $N-1$, where N is the total number of paths from s to t. If N is reasonably small (less than some threshold value), the profiler can allocate an array of counters of size N, and track path frequencies by indexing into the array using the path identifier and incrementing the corresponding counter. However, the number of potential paths in a procedure can be arbitrarily large (exponential in the number of nodes in the graph) and allocating a counter for each path can be prohibitively expensive, even infeasible in many cases. In the current example, if the threshold value is set to 4, the Ball-Larus profiler would use a hash table since there are 6 paths from s to t.

Let us assume that we are interested in profiling only a subset $I = \{sacdt, sact, sbct\}$ of paths. The Ball-Larus identifiers for the paths *sacdt*, *sact* and *sbct* are 0, 1 and 5 respectively. This means that one would have to allocate a hash table even though there are only 3 paths of interest. In such a scenario, it would be ideal if we could compute an edge assignment that allocates identifiers 0, 1 and 2 to these paths, and identifiers greater than 2 to the other paths.

Unfortunately, such an assignment is not always feasible [17]. Therefore, PPP attempts to label the edges in G such that the paths in the set I have path identifiers in $\{0, 1, 2\}$ even if such a numbering causes other (non-interesting) paths (for e.g., *sbcdt*) to share the same path identifiers. Figure 1(b) shows the

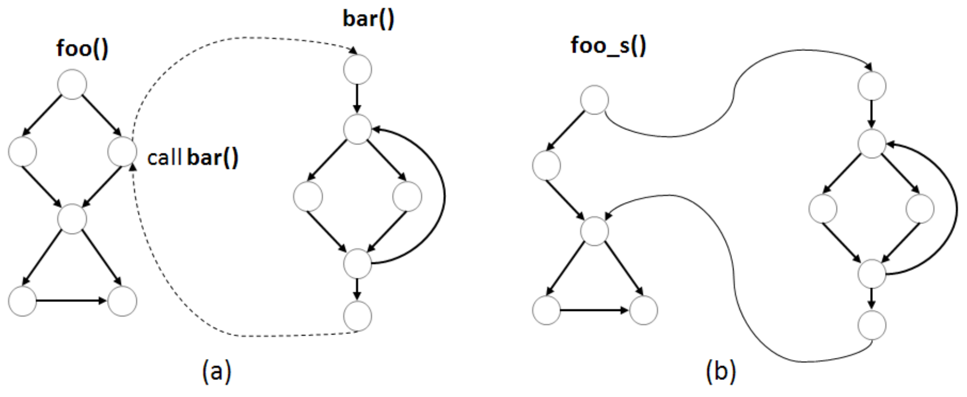


Figure 2: IPPP example. (a) Interesting interprocedural path originating in$foo()$, passes through the function $bar()$, before returning to $foo()$. (b) After function inlining, this path becomes an intraprocedural path in the supergraph $foo\_s()$.

PPP assignment of weights to edges that achieves this. This allows PPP to incur lower overheads since it can use an array to track frequencies instead of a hash table. Note that while the interesting paths *sacdt*, *sact*, *sbct* have been assigned unique identifiers from 0 to 2, the uninteresting paths *sabct* and *sbcdt* alias with the interesting paths *sacdt* and *sact* respectively. PPP resolves these aliases using Ball-Larus path identifiers, which are unique for all paths. In PPP, edges are annotated with a second weight computed using the Ball-Larus algorithm (these weights are shown in parentheses in Figure 1(c)). The profiler also stores the Ball-Larus identifiers of all interesting paths along with their counters. The occurrence of an interesting path can be detected by comparing the Ball-Larus identifier computed during the traversal with the Ball-Larus identifier stored in the array a match indicates that an interesting path was just traversed and vice versa. For example, when the uninteresting path sbcdt (PPP identifier is 2 and Ball-Larus identifier is 4) occurs, before incrementing the count at index 2 in the path array, the Ball-Larus identifier at index 2 is compared with the Ball-Larus identifier of *sbcdt*. Since they are different, the profiler infers that this path is not interesting and takes necessary action.

3 Interprocedural Preferential Path Profiling (IPPP)

Modern software development has embraced modular programming, which increases the number of procedures in a program. As a result, many interesting program behaviors span across procedure boundaries and intraprocedural testing and profiling techniques may not suffice. Unfortunately, although interprocedural analysis, profiling and testing techniques are desirable, they have traditionally been associated with high runtime overhead. For instance, interprocedural path profiling is extremely expensive [11, 15]. These high overheads have limited the use of interprocedural techniques even in laboratory testing environments where cost is usually not a prime concern. We propose to perform a simplified version of interprocedural path profiling that has low overhead. Our formulation is well suited to residual path profiling.

3.1 Preliminary Definitions

We define a few terms to facilitate the exposition of our technique.

Subpath: A subpath is an acyclic, intraprocedural path that terminates at procedure calls, in addition to loop back edges and function returns.

Whole Program Path (WPP): The whole program path is the entire sequence of subpaths generated during a given execution of a program. The WPP precisely characterizes the entire control flow behavior of the program [8].

Interprocedural Path Segment (IPS): An interprocedural path segment is a sequence of subpaths that can be generated using the following grammar

$$P \rightarrow (P \mid (P) \mid PP \mid \langle f, p\rangle$$

where '(' denotes a function call, ') represents a return, and $\langle f, p\rangle$ represents the execution of a subpath p in the function f. Intuitively, an IPS is similar to the traditional notion of interprocedurally valid paths [11], except that it does not necessarily start with a call to the main function.

3.2 Specifying Interesting IPSs

To perform interprocedural preferential path profiling, we first need to specify interesting interprocedural path segments (IPSs). While the whole program path (WPP) is a valid (though very large) IPS, for efficiency reasons we constrain the set of IPSs that can be specified. We define a depth k IPS as one that spans at most k procedure calls. For a given value of k (programmer specified), the set of IPSs exercised can be easily extracted from the WPP. For instance, consider the follow substring of a WPP:

$(\langle f1, p1\rangle(\langle f2, p2\rangle(\langle f3, p3\rangle(\langle f4, p4\rangle\langle f4, p5\rangle\langle f4, p6\rangle)\langle f3, p7\rangle$
$(\langle f5, p8\rangle)\langle f3, p9\rangle)\langle f2, p10\rangle)$

Here, the paths $p1$, $p2$, $p3$ and $p7$ terminate on procedure calls. For $k = 1$, the set of valid IPSs is

1. $\langle f1, p1\rangle(\langle f2, p2\rangle$
2. $\langle f2, p2\rangle(\langle f3, p3\rangle$
3. $\langle f3, p3\rangle(\langle f4, p4\rangle$

4. $\langle f3, p7\rangle(\langle f5, p8\rangle)\langle f3, p9\rangle$

Note that IPS 4 includes the subpath executed after returning from procedure f5. Also, IPS 3 does not extend to the subpath $\langle f4, p5\rangle$ because the path $p4$ ends at a loop back edge and our current implementation does not profile paths that span across loop boundaries.

3.3 Profiling Interesting IPSs

Much like PPP, interprocedural preferential path profiling (IPPP) achieves low overhead by exploiting knowledge about interesting paths. Given a set of interprocedural paths for which profiling information is required, we proceed in two stages. First, assuming that the number of interprocedural path segments (IPSs) that most consumers of profiles are interested in is small, we transform the interprocedural path segment profiling problem into an intraprocedural path profiling problem using function inlining. This results in a set of supergraphs that are used for analysis and instrumentation. Second, since these supergraphs are likely to contain a large number of acyclic intraprocedural paths, we use PPP to compactly number interesting paths in the supergraphs, which correspond to interesting IPSs in the original graphs. This compact numbering avoids the use of hash tables required by traditional path profiling techniques, and consequently reduces the overhead of IPS profiling. We provide an overview of IPPP with an example shown in Figure 2. Consider two functions $foo()$ and $bar()$ as shown in Figure 2(a). Assume we are interested in profiling a single IPS that originates in function $foo()$ and passes through $bar()$. In this simple scenario, the function $bar()$ is inlined into $foo()$ as shown in Figure 2(b). The inlining transformation leads to the creation of a supergraph $foo\_s()$, in which the IPS has an intraprocedural equivalent. Standard intraprocedural path profiling techniques can now be applied to such supergraphs to profile these paths. However, the supergraphs created using this function inlining transformation are likely to contain a significantly larger number of intraprocedural paths. This leads to two problems.

- First, in several cases where an array would have been sufficient for bookkeeping, path profiling will be forced to use a hash table, resulting in increased runtime overheads.
- Second, in situations where the number of paths increases beyond a threshold (for example, 2^{32}), certain paths in the graphs must be truncated leading to a loss of precision [2].

We address both of these problems. First, we use PPP to compactly number interesting paths in the supergraph. Since the number of interesting IPSs is likely to be a small subset of all possible IPSs, PPP, which provides strong guarantees about compact numbering [17], will almost always be able to use an array instead of a hash table for tracking paths. Second, we address the problem of loss of precision due to truncation by avoiding truncating edges traversed

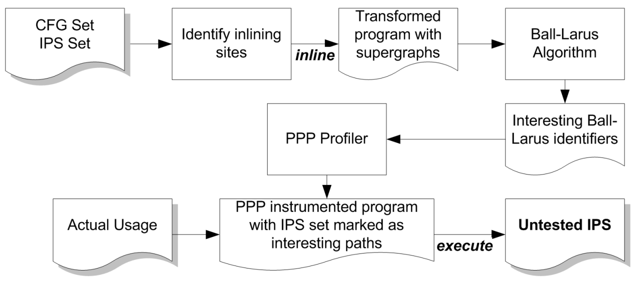

CFG Set
IPS Set
Identify inlining sites
inline
Transformed program with supergraphs
Ball-Larus Algorithm
Interesting Ball-Larus identifiers
PPP Profiler
Actual Usage
PPP instrumented program with IPS set marked as interesting paths
execute
Untested IPS

Figure 3: Overview of the interprocedural preferential path profiling scheme.

by interesting IPSs. Consequently, interesting IPSs are profiled precisely and efficiently.

3.4 IPPP Algorithm

This section informally describes our algorithm for profiling interesting interprocedural path segments (IPSs). Figure 3 provides an overview of the technique.

1. Input: A set of CFGs and a set S of interesting depth k-limited IPSs.

2. Identify inlining sites: Based on the paths in set S, identify the set of call sites for inlining. For each IPS $\langle f1, p1\rangle(\langle f2, p2\rangle, \ldots$, consider all the subpaths that terminate at a procedure call. All such procedure call sites are identified and marked.

3. Mark all edges traversed by IPSs: Assign a globally unique identifier to all IPSs. Traverse all edges along each IPS and mark the edges with the corresponding IPS identifier. Edges that participate in multiple IPSs will have multiple IPS identifiers associated with them. This labelling serves two purposes. First, it helps create a mapping between an IPS in the original collection of CFGs and its corresponding intraprocedural equivalent in the transformed supergraph (see step 6). Second, it marks these edges as non-candidates for truncation, if truncation is necessary (see step 5).

4. Supergraph construction: Create supergraphs as shown in Figure 2(b) by combining the CFGs of individual procedures as determined in step 2.

5. Ball-Larus numbering: Assign Ball-Larus numbers to all paths in each supergraph. Ensure that edges traversed by IPSs as marked in step 3 are not truncated. After this step, each path in a supergraph is assigned a unique identifier.

6. Identify interesting IPSs: From the Ball-Larus numbering and the IPS edge information computed in step 3, obtain the Ball-Larus identifiers of interesting IPSs.

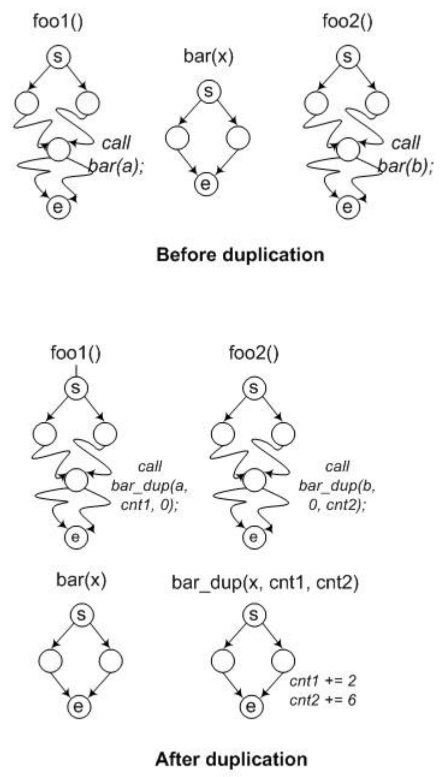


Figure 4: An example that illustrates an IPPP approach based on function replication that avoids increase in code size due to inlining-based IPPP.

7. Drive PPP: Use the Ball-Larus identifier of interesting IPSs computed in Step 6 as input to the PPP algorithm.

3.5 Avoiding Inlining through Code Duplication

IPPP uses procedure inlining to convert an interprocedural path profiling problem to an intraprocedural one. In IPPP, each procedure is effectively duplicated as many times as the number of unique contexts it occurs in the set of input IPSs. However, this approach may not scale if the number and/or depth of IPSs in the input specification is large or the IPS passes through procedures with large bodies. While our current implementation uses selective inlining, an alternative is to duplicate the code.

We now describe the approach based on procedure duplication using an example. Consider the program in Figure 4. The program consists of three functions, $foo1()$, $foo2()$ and $bar(x)$. Both $foo1()$ and $foo2()$ have calls to $bar(x)$. Assume that the input to IPPP consists of two interprocedural path segments that pass through both call sites. As per the approach described in the previous section, we would inline a copy of $bar(x)$ in $foo1()$ and in $foo2()$ (the original copy of $bar(x)$ remains). If the resulting increase in code size is not acceptable, IPPP creates a second copy of the procedure $bar(x)$ (say $bar\_dup(x)$) and shares it across multiple contexts as shown in Figure 4. Here, we only inline the procedure bar into $foo1()$ and $foo2()$ for the purpose of analysis and compute an assignment of weights to the edges of $bar\_dup(x)$ independently in each context. We then map these weights back to the original edges of $bar\_dup(x)$. Each edge of $bar\_dup(x)$ is now associated with as many weights as the number of unique contexts (say N). Next, we add N counter variables as arguments to the procedure $bar\_dup(x)$, one for each context, to obtain $bar\_dup(x, cnt1, cnt2, \ldots, cntN)$. We also modify the respective call sites in each context to pass the current value of the Ball-Larus and/or PPP counter as an argument to the duplicated procedure, as shown in Figure 4. The edges of $bar\_dup(x, cnt1, cnt2)$ are instrumented to increment all counter variables as and when required. These increment operations are usually inexpensive and are unlikely to have an adverse effect on performance, at least for a moderate number of contexts. Finally, before returning from the procedure $bar\_dup(x, cnt1, cnt2, \ldots, cntN)$, we save the state of all counter variables so that they can be retrieved by the respective callers. In this way, both the Ball-Larus and PPP algorithms can be extended to the interprocedural case without an excessive increase in code size.

3.6 Discussion

The key enabling insight that allows inlining/code duplication to avoid exponential blowup issues is they are based off concrete Ball-Larus path-profiles from a regression test suite that typically exercises only a small fraction of all possible interprocedural-paths. Our experiments (Section 5) show that the code size increase for IPPP is only 22% higher on average than standard intra-procedural Ball-Larus PP. Recursion is handled as we are only interested in IPSs of limited depth. Function pointers are handled by doing context-sensitive inlining from concrete Ball-Larus path profiles and using runtime-checks for validation.

4 Residual Path Profiling (RPP)

Residual path profiling identifies the set of paths executed by deployed software that were not tested during software development [14]. This section describes how we perform residual profiling for intraprocedural and interprocedural paths.

4.1 Intraprocedural Residual Path Profiling

RPP for intraprocedural paths is fairly straightforward and proceeds in two stages. First, a program is instrumented for path profiling with the Ball-Larus technique and run with its test suite inputs. The Ball-Larus identifiers of intraprocedural paths that were executed are recorded. Next, the same program is instrumented with PPP with the recorded Ball-Larus paths marked as interesting paths and this version of the program is deployed to gather real usage profiles, perhaps as part of a beta testing phase. When an untested path is executed, either its PPP identifier will exceed the size of the array used to track paths or its Ball-Larus identifier will not match the one recorded in the array entry (see Figure 1(c)). In this way, untested paths are detected and recorded during actual usage of deployed software.

4.2 Interprocedural Residual Path Profiling

We can use the interprocedural preferential path profiling (IPPP) technique described in Section 3 to perform residual profiling of interprocedural paths. First, we need to specify the set of interesting IPSs that should be profiled to the IPPP algorithm. This is done by performing whole program path (WPP) profiling on the test suite inputs. Given a user specified value for k, we can identify all exercised depth-k constrained IPSs from the WPP. These depth-k IPSs are used as input to the IPPP algorithm. This generates a new binary ready for deployment. Running this binary produces a list of paths that are exercised in the current run but were not exercised by the test suite. The Ball-Larus identifiers of these untested paths can be used to generate the set of untested IPSs [2].

4.2.1 Alternative scheme for Interprocedural Residual Path Profiling

An alternative technique that does not require the use of WPP profiles is illustrated in Figure 5. Our experimental results reported in Section 5 use this technique for interprocedural residual path profiling. The scheme proceeds in three stages.

Stage I: The program is profiled on its test suite inputs to identify all acyclic, intraprocedural paths that were exercised (Figure 5(a)). The Ball-Larus technique is used to perform path profiling.

Stage II: The path profile from Stage I is used to identify all call sites that can occur on a depth-k IPS. Note that this is an overapproximation of the set of call sites that would have been identified given a WPP profile. These call sites are inlined and a new binary, instrumented to collect Ball-Larus path profiles, is generated. As a result of inlining, all depth-k IPSs exercised by the test suite inputs appear as intraprocedural paths in the new binary. This binary is then instrumented to collect Ball-Larus path profiles and rerun on the test suite inputs. The path profile generated effectively assigns unique Ball-Larus identifiers to all depth-k IPSs exercised. Figure 5(b) illustrates this process.

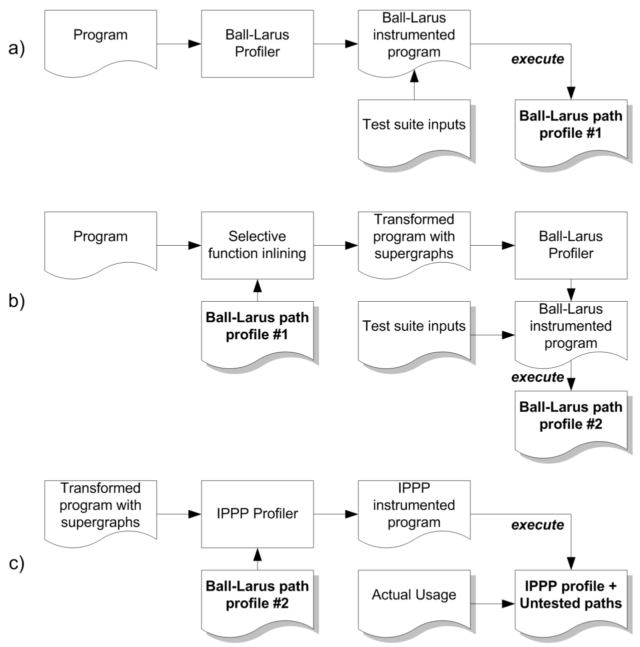

a)
Program
Ball-Larus Profiler
Ball-Larus instrumented program
execute
Test suite inputs
Ball-Larus path profile #1
b)
Program
Selective function inlining
Transformed program with supergraphs
Ball-Larus Profiler
Ball-Larus path profile #1
Test suite inputs
Ball-Larus instrumented program
execute
Ball-Larus path profile #2
c)
Transformed program with supergraphs
IPPP Profiler
IPPP instrumented program
execute
Ball-Larus path profile #2
Actual Usage
IPPP profile + Untested paths

Figure 5: Residual path profiling for IPPPs. (a) Profile executed paths on test suite inputs using the Ball-Larus scheme. (b) Inline functions to convert executed IPSs to intraprocedural paths and re-execute on test inputs. (c) Use path profile from (b) to drive PPP and run program with field inputs to detect untested paths.

Stage III: The path profile generated in the previous step along with the transformed binary serve as inputs to the PPP algorithm as shown in Figure 5(c). This generates a new binary suitable for deployment, where all depth-k IPSs that were exercised by the test suite inputs are marked as interesting paths. When this binary is executed on field inputs the untested paths reported correspond to depth-k IPSs (and intraprocedural paths) not exercised by the test suite.

4.3 Discussion

Testing large software is a resource constrained activity. Consequently, the priority is to test most frequently exercised behaviors across all users (not a single user). RPP accomplishes this by recording all untested paths exercised. This information can be aggregated across users to determine priorities. A scheme that only records frequent untested paths would miss rare untested paths that all/most users execute.

Benchmark	#untested paths	%untested paths	%freq of untested paths	#funcs with untested paths	#untested edges	#funcs with untested edges	#untested paths in funcs with no untested edges	%untested paths in funcs with no untested edges
164.gzip	80	7.2	0.0	6	3	2	77	96.3
175.vpr	274	20.9	0.0	29	147	22	13	4.7
179.art	132	50.0	47.2	12	130	10	6	4.5
181.mcf	3	1.2	0.0	3	8	1	2	66.7
183.equake	1	0.5	0.0	1	0	0	1	100
188.ammp	117	22.5	0.0	4	2	1	114	97.4
197.parser	612	13.0	0.0	61	211	29	273	44.6
256.bzip2	398	45.1	0.0	13	81	10	26	6.5
300.twolf	295	11.3	0.0	36	43	18	117	39.7
PCgame-1	970	19.8	1.5	139	248	81	502	51.8
PCgame-2	3531	16.5	0.6	248	384	143	898	25.4
Average	583	18.9	4.5	50.2	114.3	28.8	184.5	48.9

Table 1: Untested intra-procedural path information obtained from residual path profiling.

5 Experiments

We have implemented our techniques (intra and interprocedural PPP) using the Scale compiler infrastructure [10]. In addition, we have implemented intraprocedural PPP using Microsoft's Phoenix compiler and are currently working on a Phoenix implementation of interprocedural PPP. We use benchmarks from the SPEC CPU2000 suite for evaluation as well as a couple of large (1-2 million LOC), resource-intensive PC games. The PC games were compiled using Microsoft's Phoenix compiler. The SPEC benchmarks were run to completion on an Alpha 21264 processor running Digital OSF 4.0 and the PC games were run on a 2 GHz Intel x86 processor running Microsoft Vista. All timing numbers reported use the hardware cycle counter. We report overhead numbers for the games using frames per second as the performance metric.

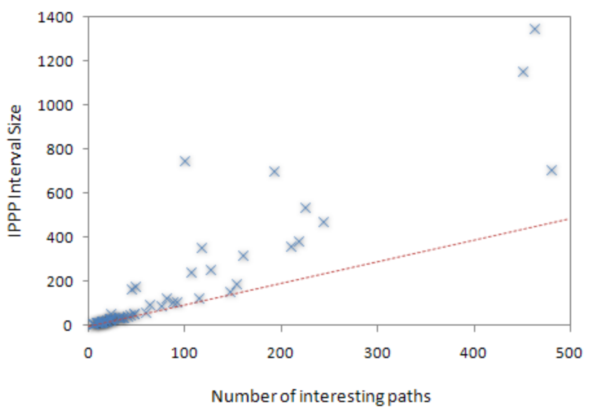


Figure 6: Compact numbering achieved by IPPP for several functions from programs in the SPEC CPU2000 suite.

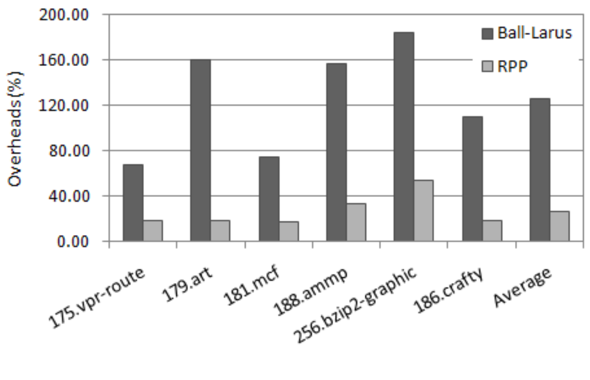


Figure 7: Runtime overheads of interprocedural path profiling.

5.1 Interprocedural Preferential Path Profiling (IPPP)

To evaluate our IPPP scheme, we labeled all depth 1 interprocedural path segments (i.e., all IPSs that span a single procedure call boundary) exercised by the SPEC train inputs as interesting IPSs and produced an instrumented binary that profiles these interprocedural paths. This binary was then run on the ref inputs. Since we perform selective inlining to convert IPSs into intraprocedural paths, we can also profile these paths with the Ball-Larus technique.

Figure 7 shows the overhead of these techniques on some of the SPEC benchmarks (the Scale compiler does not successfully compile all SPEC benchmarks, even without our path profiling). The Ball-Larus scheme incurs high overheads that range from 70% to 180% with an average of 125%. With the exception of `256.bzip2`, which incurs an overhead of 52%, IPPP achieves reasonably low overhead with an average of 26%. For four of the six benchmarks the overhead is less than 20%. This overhead may be low enough to permit residual profiling of IPSs during beta software testing of many interactive desktop applications, such as web browers, email clients, and productivity software, where the slowdown is possibly below human perception threshold on fast modern machines.

IPPP is able to achieve significantly lower overheads as compared to interprocedural Ball-Larus because it is able to compactly number interesting IPSs and avoid using a hash table for recording path information. Figure 6, which plots the size of the interval allocated to interesting paths versus the number of interesting paths for functions in the SPEC benchmarks, substantiates this claim. For almost all procedures this ratio is very close to 1, indicating almost perfect compaction.

5.2 Intraprocedural Residual Path Profiling

We used the train and ref inputs provided with the SPEC benchmarks to approximate a residual profiling scenario. For the PC games, we played the games

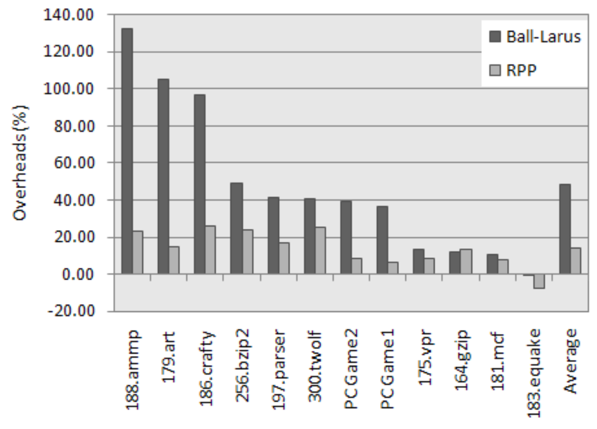


Figure 8: Comparison of runtime overheads of Ball-Larus and intraprocedural RPP.

for different lengths of time (5 minutes and 10 minutes). As the frame rate reduction during intraprocedural RPP was small as indicated in Figure 8, we were able to play both games as we would if they were not being profiled. First, the Ball-Larus path profiler was used to collect profiles of the benchmarks on their train inputs. The paths profiled become the interesting paths that are input to the preferential path profiler (PPP). PPP generated a new instrumented binary which was then run on the benchmarks reference input. Paths that were exercised only by the ref input are reported as untested paths. We also performed residual edge profiling and compare the results against RPP. The results are shown in Table 1. Many of the SPEC benchmark ref inputs are merely larger size versions of the train inputs. Despite this, the data indicates that the ref inputs exercise many more paths. The execution frequency contribution of these paths show that, with the exception of `179.art`, which exercised new hot paths, these untested paths are rarely executed. The comparison between untested paths and untested edges is striking. For these benchmarks, a significant number of untested paths occurred in functions which reported no new untested edges. These paths would go undetected with residual edge profiling and demonstrate the advantage of RPP. Even for untested paths that exercise new edges, inferring the path from the edge profile is not always possible. The PC games, which are significantly larger than the SPEC benchmarks, show similar results. For both games, the longer scenario exercises a large number of rarely executed untested (by the shorter scenario) paths. Many of these paths do not include any new edges.

Figure 8 reports the overhead of performing intraprocedural RPP for the scenario described above relative to Ball-Larus profiling.The overhead numbers for the PC games represent percentage reduction in frame rate as a result of

Benchmark	#untested paths	%untested paths	%freq of untested paths
175.vpr	300	21.1	0.0
179.art	199	77.4	57.5
181.mcf	3	1.1	0.0
183.crafty	3262	63.5	0.0
188.ammp	123	21.5	1.4
256.bzip2	949	58.3	0.1
Average	314.8	35.9	11.8

Table 2: Untested IPS information obtained from interprocedural residual path profiling.

profiling. The overhead numbers (average of 13%) are in line with those reported in [17] (average of 20%). The numbers are lower due to a more optimized implementation and because we use train inputs to generate interesting paths while their experiments used the *ref* inputs for this purpose. In addition, the new benchmarks (PC games) incur lower overhead (7-9%). The numbers for the PC games are especially impressive as these are cutting-edge, resource intensive programs and indicate that RPP can be used in deployed software, at least during beta software testing.

5.3 Interprocedural Residual Path Profiling

We performed a similar experiment to that described in Section 5.2, except that we labelled all depth-1 interprocedural path segments exercised by the SPEC train inputs as interesting IPSs and then ran the IPPP generated binary on the benchmarks ref input. Table 2 reports the results. Comparing entries from Table 2 with Table 1 for a few of the benchmarks, it can be seen that IPPP exposes a larger number of untested paths.

5.4 Residual Path Profiling Simulation

We performed an experiment much like the one described in Section 5.2 for residual path profiling of intraprocedural path except that we use a larger number of train and ref inputs to simulate a residual profiling scenario. We use the MinneSPEC input suite along with the SPEC test and train inputs as representative of the test suite. The results are shown in Table 3. It is interesting to note that even though a large number of paths are exercised by the train inputs for these programs, we are able to detect a significant number of new paths on the ref inputs which essentially characterize the “deployed” behaviours of these programs.

Benchmark	#untested paths	%untested paths	%freq of untested paths
176.gcc-200	2670	4.6	0.3
176.gcc-scilab	2635	4.5	0.3
176.gcc-expr	723	1.2	0.0
179.gcc-166	2034	3.5	0.0
179.gcc-integrate	238	0.4	0.0
256.bzip2-graphic	249	6.1	0.0
256.bzip2-source	520	12.8	0.0
256.bzip2-program	49	1.2	0.0
175.vpr-place	175	1.6	0.0
175.vpr-route	30	0.3	0.0
Average	932	3.6	0.1

Table 3: Untested intra-procedural path information obtained using a more robust test suite.

5.5 Code Size Increase

Apart from the runtime overheads of tracking paths, path profiling schemes (Ball-Larus and PPP) also increase the size of the program binary. The reasons for the code bloat are two-fold: (a) instrumentation placed along edges of functions and at the end of every path, (b) space allocated to path counter tables and auxillary structures. Table 4 shows the increase in code size (number of times relative to the unprofiled binary) caused by both the intra-procedural profiling schemes for a set of benchmarks programs. On average, the Ball-Larus profiling scheme increase the code size by a factor of 3.21. The increase in code size due to PPP is slightly lower at 3.03, primarily because a more compact numbering reduces the amount of space that must be allocated for the path counter tables.

We also measured the increase in code size caused by the interprocedural versions of the path profiling schemes. One might have anticipated a significant increase in code size compared to intraprocedural profiling because of inlining. As shown in Table 5, this turns out to be true for Ball-Larus interprocedural profiling scheme, which increases code size by a factor of 5.33. However, the code bloat due to IPPP is significantly lower because inlining is restricted to call-sites along a small set of interesting paths. The average increase in code size is 4.35, which is close to the increase in code size due to intraprocedural path profiling. These results show that the increase in code size due to IPPP is much lower than expected and does not limit the applicability of interprocedural path profiling any more than the intraprocedural profiling schemes.

Benchmark	Ball-Larus	PPP
188.ammp	3.25	3.40
179.art	4.57	4.71
186.crafty	3.67	2.83
256.bzip2	2.60	2.40
197.parser	2.26	1.71
300.twolf	2.71	2.52
PC Game 1	5.41	5.00
PC Game 2	4.60	4.57
175.vpr	3.38	3.38
164.gzip	3.27	3.22
181.mcf	1.23	1.20
183.equake	1.56	1.44
Average	3.21	3.03

Table 4: Increase in code size due to Ball-Larus and PPP intraprocedural profiling.

Benchmark	Ball-Larus	IPPP
175.vpr	6.91	5.00
179.art	4.83	3.67
181.mcf	4.40	3.80
188.ammp	5.52	4.81
256.bzip2	6.62	5.38
186.crafty	3.71	3.44
Average	5.33	4.35

Table 5: Increase in code size due to Ball-Larus and interprocedural path profiling.

6 Related Work

Melski and Reps extended Ball-Larus profiling to capture interprocedural paths [11]. They create a single supergraph that connects all procedures, and then apply the Ball-Larus numbering to label paths in this graph. Tallam et al. proposed a technique to profile overlapping path fragments from which interprocedural and cyclic paths can be estimated [14]. Both these techniques have considerably higher overhead than the Ball-Larus technique for profiling intraprocedural, acyclic paths as well as our technique. Our scheme for profiling interprocedural paths achieves low overhead by exploiting the observation that in many cases the interprocedural paths of interest are small in number, can be compactly encoded, and are known in advance.

Researchers have recognized the importance of using field data to improve testing [6, 13]. To limit profiling overhead they have focused on collecting inexpensive function/node/
edge coverage profiles, using sampling, or using dynamic instrumentation. However, as we have shown, path profiles provide richer information than edge profiles, at least for residual testing. The Gamma project [13] samples across

program instances by splitting monitoring tasks across different instances of the software. This enables partial information to be collected from different users with low-overhead and later integrated to gather overall profiles. Liblit et al. [9] also rely on instrumenting a large number of instances of the same software. They randomly sample each instance at a low rate to achieve small overheads and collate information from multiple users to isolate bugs. Residual path profiling could use these techniques to further reduce overhead. However, these techniques require a fair amount of infrastructure and the cooperation of a large numbers of users to be successful. Residual path profiling is able to achieve low overhead on a single program instance without this requirement. Arnold and Ryder [1] proposed a sampling scheme based on code duplication scheme that creates an instrumented and non-instrumented version of each function and switches between these versions. They achieve low overhead by controlling the sampling rate and ensuring that the non-instrumented function version is executed most of the time. This technique is well-suited to performance profiling but would miss rarely executed paths during residual testing. Chilimbi and Hauswirth [5] extended this scheme with an adaptive sampling technique that can capture rare events while still incurring low overhead. However, their granularity of adaptive sampling is a code fragment that starts and ends at a function entry point or loop backward branch. These code fragments can potentially contain many paths. If any one of the paths is hot, the fragment will be rarely sampled and infrequently exercised code paths in the same fragment will go unprofiled.

Tikir and Hollingsworth dynamically insert instrumentation on method invocation for node coverage [16]. A separate thread periodically removes the instrumentation via a garbage collection process. They report an average overhead of 36% for C programs. Similarly, the Jazz tool [12] dynamically instruments for node coverage and def-use coverage. They use a test planner to remove instrumentation when it is no longer needed. They report impressive overheads of 3% on average for node coverage of Java programs. Their overheads for def-use coverage are much higher (average of 127%). Unfortunately, such dynamic instrumentation techniques cannot be applied to reduce the overhead of path profiling as the path counter updates cannot be removed. To the best of our knowledge, *we are the first to implement and evaluate a practical scheme for residual path profiling.* Prior work on residual testing has focused on node coverage [14]. Node coverage information is much cheaper to collect, but contains less information than edge profiles, which we show are inferior to path profiles.

7 Conclusions

Software testing is extensively used for uncovering bugs in large, complex software. However, test suites are typically designed with little information about actual software usage. We have shown how recent advances in profiling program paths with low-overhead has provided the opportunity to perform residual path profiling on deployed software. This identifies all paths executed by deployed software that were untested during software development. This information can be used to improve regression test suites used for unit testing, where individual software modules are tested in isolation. We have extended our low-overhead path profiling technique to capture interprocedural paths. Residual interprocedural path profiles are useful for improving integration testing, where groups of modules are tested together. Our experimental results show that low-overhead path profiling, both intraprocedural and interprocedural, provides valuable quantitative information on testing effectiveness. We show that residual edge profiling is inadequate as a significant number of untested paths include no new untested edges.

In addition, our low-overhead path profiling techniques can be applied to perform more informed compiler optimization driven by real-usage path-profiles. These profiles can be used to check hotness assumptions made by in-house optimization. They could also be used for security applications where certain program behaviors need to be restricted based on runtime criteria.

References

[1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI)*, pages 168–179, 2001.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture (MICRO)*, pages 46–57, 1996.

[3] B. Beiser. *Software testing techniques*. Van Nostram Reinhold Inc., N. Y., 1990.

[4] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *International Symposium on Code Generation and Optimization (CGO)*, pages 205–216, 2005.

[5] T. M. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 156–164, 2004.

[6] M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering.*, 31(4):312–327, 2005.

[7] R. Joshi, M. D. Bond, and C. B. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *International Symposium on Code Generation and Optimization (CGO)*, pages 239–250, 2004.

[8] J. R. Larus. Whole program paths. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 259–269, 1999.

[9] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI)*, pages 141–154, 2003.

[10] K. S. McKinley, J. Burrill, M. D. Bond, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z.Wang, and C. Weems. The Scale compiler. `http://ali-www.cs.umass.edu/Scale`, 2005.

[11] D. Melski and T. W. Reps. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction (CC)*, pages 47–62, 1999.

[12] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *Proceedings of the 27th international conference on Software engineering (ICSE)*, pages 156–165, 2005.

[13] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 11th ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, pages 128–137, 2003.

[14] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st international conference on Software engineering (ICSE)*, pages 277–284, 1999.

[15] S. Tallam, X. Zhang, and R. Gupta. Extending path profiling across loop backedges and procedure boundaries. In *International Symposium on Code Generation and Optimization (CGO)*, pages 251–264, 2004.

[16] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA)*, pages 86–96, 2002.

[17] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *ACM SIGPLAN Conference on Principles of Programming Languages (POPL)*, pages 351–362, 2007.