Quantifying the Effectiveness of Testing via Efficient Residual Path Profiling

Trishul M. Chilimbi
Microsoft Research Redmond
trishulc@microsoft.com

Aditya V. Nori
Microsoft Research India
adityan@microsoft.com

Kapil Vaswani
Indian Institute of Science, Bangalore
kapil@csa.iisc.ernet.in

ABSTRACT

Software testing is extensively used for uncovering bugs in large, complex software. Testing relies on well designed regression test suites that anticipate all reasonable software usage scenarios. Unfortunately, testers today have no way of knowing how much of real-world software usage was untested by their regression 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.

Categories and Subject Descriptors: D.2.5 [Testing and Debugging]: Tracing

General Terms: measurement, reliability

Keywords: path profiling, testing, residual, inter-procedural

1. INTRODUCTION

Proving programs correct is hard. Static analysis tools attempt to address this problem by checking that a program satisfies a set of specifications on all possible execution paths. However, these tools 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, 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 in a variety of environments. Ideally, one would like to profile actual software usage, perhaps during beta testing, to detect untested behaviors.

Unfortunately, collecting path profiles using the standard Ball-Larus path profiling technique [1] is expensive. Recently, researchers have proposed a technique called preferential path profiling (PPP) that reduces the overhead of path profiling even when the number of paths is large [9]. In contrast to the Ball-Larus profiling scheme, which assigns weights to edges of a control flow graph (CFG) such that all paths are allocated unique identifiers [1], PPP assigns weights to the edges such that the sum of the weights along the edges of the a given subset of 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.

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 [7]. 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 [2], which combines individual modules and tests them as a group, as this requires profiling interprocedural paths. 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.

2. 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 [6, 8]. These high overheads have limited the use of interprocedural techniques even in laboratory testing environments where cost is usu-

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
ACM 978-1-59593-812-1/07/0009.

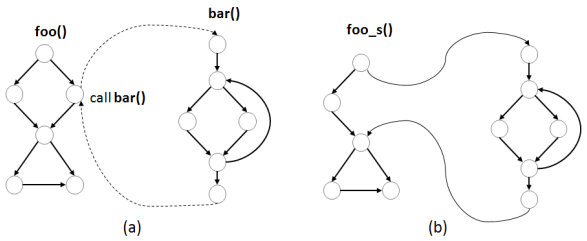

Figure 1: IPPP example. (a) Interesting interprocedural path originating in$foo()$, passes through the function $bar()$, before returning to $foo()$. (b) After function inlining, this path becomes an intraprocedural path in the supergraph $foo\_s()$.

ally not a prime concern. We propose to perform a simplified version of interprocedural path profiling that has low overhead and is well suited to residual profiling.

2.1 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 [4].

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. We also define a depth-k IPS is an IPS that spans at most k procedure calls. The profiling scheme we propose extends PPP to work with depth-k IPS (for a programmer-specified k) rather than intraprocedural paths.

2.2 Profiling Interesting IPSs

This section describes our algorithm for profiling interesting interprocedural path segments (IPSs). We first provide an overview of IPPP with an example shown in Figure 1. Consider two functions $foo()$ and $bar()$ as shown in Figure 1(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 1(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 function inlining are likely to contain a significantly larger number of intraprocedural paths, forcing a traditional path profiler to use hash tables. We address this problem by using 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 (in this example, only one IPS is interesting), PPP will almost always be able to use an array instead of a hash table for tracking paths, reducing runtime overheads significantly.

The algorithm for profiling interesting interprocedural path segments proceeds as follows.

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 [1], if truncation is necessary (see step 5).
4. Supergraph construction: Create supergraphs as shown in Figure 1(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.
7. Drive PPP: Use the Ball-Larus identifier of interesting IPSs computed in Step 6 as input to the PPP algorithm.

3. RESIDUAL PATH PROFILING

Residual path profiling identifies the set of paths executed by deployed software that were not tested during software development [7]. This section describes how we perform residual profiling for intraprocedural and interprocedural paths.

RPP for intraprocedural paths is fairly straightforward. The set of paths exercised by inputs in a test suite are recorded using a Ball-Larus profiler and fed as input to PPP, which generated a new instrumented binary. This version of the program is deployed to gather real usage profiles, perhaps as part of beta testing. When an untested path is executed, PPP simply logs the path identifer of the untested path.

We can use the interprocedural preferential path profiling (IPPP) technique described in Section 2.2 to perform residual profiling of interprocedural paths. First, we need

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.

to specify the set of interesting IPSs that should be profiled to the IPPP algorithm. This is done by performing whole program path (WPP) [4] profiling on the test suite inputs. For a given value of k (programmer specified), the set of IPSs exercised in a given run of the program can be easily extracted from the WPP. We also propose an alternate approach for extracting exercised IPS that does not require the WPP; details of this approach can be found in [3].

4. EXPERIMENTS

We have implemented our techniques (intra and interprocedural PPP) using the Scale compiler infrastructure [5]. 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. We report overhead numbers for the games using frames per second as the performance metric.

4.1 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 for different lengths of time (5 minutes and 10 minutes). The results are shown in Table 1. While performing residual profiling intra-procedurally, we find that the ref inputs exercise many more paths than the train inputs. The frequency distribution of paths shows that untested paths are rarely executed. Furthermore, 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. We also find that the overhead numbers (average 13%) are in line with those reported in [9] (average 20%). The numbers for the PC games (average 7-9% overhead) 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 testing.

4.2 Interprocedural Residual Path Profiling

We performed a similar experiment to that described in Section 4.1, 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

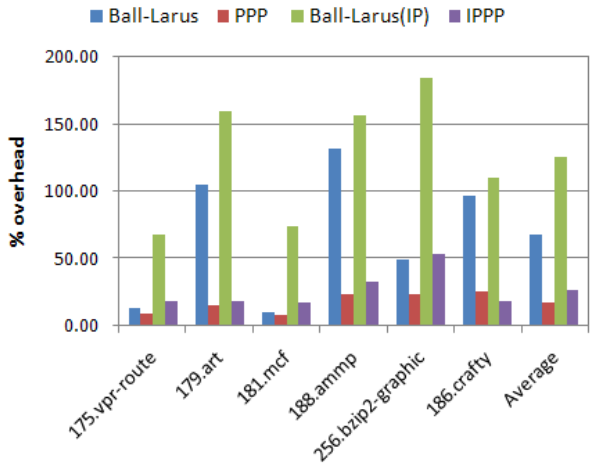

Figure 2: Runtime overheads of interprocedural path profiling.

the benchmarks ref input. Since we perform selective inlining to convert IPSs into intraprocedural paths, we can also profile these paths with the Ball-Larus technique. Figure 2 shows the overhead of this technique for 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% because it is able to compactly number interesting IPSs and avoid using a hash table for recording path information. This overhead may be low enough to permit residual profiling of IPSs during beta software testing. Table 2 characterizes the residual paths detected by IPPP. The results show that IPPP exposes a much larger number of untested paths as compared to PPP.

4.3 Residual Path Profiling Simulation

We performed an experiment much like the one described in Section 4.1 for residual path profiling of intraprocedural path except that we use a larger number of train and ref

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.

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.

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.

4.4 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. Our experiments for set of benchmarks programs from the SPEC CPU2000 suite show that on average, the Ball-Larus profiling scheme increase the code size by a factor of 3.21 on average. 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. This turns out to be true for Ball-Larus interprocedural profiling scheme, which increases code size by a factor of 5.33 on average. However, the code bloat due to IPPP is significantly lower (average 4.35) because inlining is restricted to call-sites along a small set of interesting paths. 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.

5. RELATED WORK

Melski and Reps extended Ball-Larus profiling to capture interprocedural paths [6]. 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. Prior work on residual testing has focused on node coverage [7]. Node coverage information is much cheaper to collect, but contains less information than edge profiles, which we show are inferior to path profiles.

6. CONCLUSIONS

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 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.

7. REFERENCES

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

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

[3] T. Chilimbi, A. V. Nori, and K. Vaswani. Quantifying the effectiveness of testing via efficient residual path profiling. Technical Report TR-2007-62, Microsoft Research, 2007.

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

[5] 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.

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

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

[8] 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.

[9] 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.