5

Related Work

Given enough eyes, all bugs are shallow.

–Eric S. Raymond, The Cathedral and the Bazaar

Here we discuss a cross section of related work, loosely organized into three broad topics. We briefly visit static analyses that examine code without running it. We consider earlier approaches to profiling and tracing running code, most of which have concentrated on performance profiling. Lastly we review dynamic analyses that focus more directly on the problem of debugging, including several that use statistical methods.

5.1 Static Analysis

There is currently a great deal of interest in applying specialized static analysis to improve software quality. Static analyses can help to find bugs earlier in development when they are cheaper to fix. Purely analytic approaches build upon such formalisms as type systems [25, 37], automated theorem proving [22], and software model checking [31]. Some static analyses can guarantee that certain classes of bugs can never occur in any run. Strong assurances of this form may be required in certain domains, and cannot generally be obtained from dynamic schemes such as CBI.

While we firmly believe in the use of static analysis to find and prevent bugs, our dynamic approach has advantages as well. A dynamic analysis can observe actual run-time values, which is often better than either making a very conservative static assumption about run-time values for the sake of soundness, or allowing some very simple bugs to escape undetected. Another advantage of dynamic analysis, especially one that mines actual user executions for its data, is the ability to assign an accurate importance to each bug. Additionally, as we have shown, a dynamic analysis that does not require an explicit specification of the properties to check can find clues to a very wide range of errors, including classes of errors not considered in the design of the analysis.

A complementary family of static bug hunting tools places more emphasis on human factors in software development. Static metrics of software complexity and other factors can guide engineers to likely hiding places for bugs [51]. The chronological record captured by a source code control system can be mined to reveal areas

B. Liblit: Cooperative Bug Isolation, LNCS 4440, pp. 89–93, 2007.

of high code churn or to raise warning flags when code is changed in a manner inconsistent with historical patterns [63, 66]. Any of these systems might reasonably be integrated with Cooperative Bug Isolation, such as by increasing the sampling density in code that appears suspect.

5.2 Profiling and Tracing

Dynamic program sampling has a long history, with most applications focusing on performance profiling and optimization. Any sampling system must define a trigger mechanism that signals when a sample is to be taken. Typical triggers include periodic hardware timers or interrupts [9, 59, 61], periodic software event counters (e.g., every *n*th function call) [3], or a hardware/software mix. In most cases, the sampling interval is strictly periodic. Periodic sampling may suffice when hunting for large performance bottlenecks, but may systematically miss rare events.

The Digital Continuous Profiling Infrastructure [1] is unusual in choosing sampling intervals randomly. However, the random distribution is uniform, such as one sample every 60K to 64K cycles. Samples thus extracted are not independent. If one sample is taken, there is zero chance of taking any sample in the next 1–59,999 cycles and zero chance of *not* taking exactly one sample in the next 60K–64K cycles. We trigger samples based on a geometric distribution, which correctly models the interval between successful independent coin tosses. The resulting data is a statistically rigorous fair random sample, which in turn grants access to a large domain of powerful statistical analyses.

Recent work in trace collection has focused on program understanding. Techniques for capturing program traces confront challenges similar to those we face here, such as minimizing performance overhead and managing large quantities of captured data. Dynamic analysis in particular must encode, compress, or otherwise reduce an incoming trace stream in real time, as the program runs [14, 55]. It may be difficult to directly adapt dynamic trace analysis techniques to a domain where the trace is sampled and therefore incomplete.

Path profiling subsumes basic block and edge profiling to directly monitor how often each acyclic path in a program executes. Optimizations first developed by Ball and Larus [4] limit instrumentation to loop back edges and chord edges not in a special spanning tree constructed for each acyclic region. This work relates to ours in three respects. First, feedback reports containing path profiles may be an interesting target for bug isolation data mining. A program may succeed on most paths but fail on some others. Second, the acyclic regions we use for overhead amortization (Sect. 2.1.1) correspond well to those used by Ball and Larus to build spanning trees. It should not be difficult to integrate the two analyses, thereby collecting fair random samples of complete path profiles. Lastly, a variant of our proposed path balancing algorithm (Sect. 2.3.6) might be layered atop a path profiler. Paths within an acyclic region carry unique indexes. Instead of balancing all paths, a path-indexed table could record the exact (unbalanced) instrumentation weight for each path. This