Sampling User Executions for Bug Isolation

Ben Liblit Alex Aiken Alice X. Zheng Michael I. Jordan
{liblit,aiken,alicez,jordan}@cs.berkeley.edu
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720-1776

1. Introduction

Many computer scientists think of a program as either correct (i.e. it meets some specification) or incorrect (i.e. it does not meet some specification). But industrial software development is as much about economics as computer science. Software quality is a monetary balancing act among engineers' salaries, time to market, user expectations, and other business concerns. We ship software when it seems correct enough to neither embarrass us nor alienate users. We ship software with known bugs that are not worth fixing, and users uncover new bugs that we never imagined.

Practitioners clearly need something other than a Boolean notion of correctness, but such a notion has been difficult to quantify. In-house testing can only guess at field usage patterns, and poor guesses can leave users in bad shape. An obscure, low-priority bug that was difficult to reproduce in the testing lab may turn out to affect large numbers of users on a regular basis. Technical support channels provide one way for post-deployment feedback to reach engineers, but traditionally these mechanisms have been informal and overly dependent on human intervention.

Widespread Internet connectivity makes possible a radical change to this situation. For the first time it is feasible to directly observe the reality of a software system's deployment. Through sheer numbers, the user community brings far more resources to bear on exercising a piece of software than could possibly be provided by the software's authors. Coupled with an instrumentation and reporting infrastructure, these users can potentially replace guesswork with real triage, directing scarce engineering resources to those areas that benefit the most people.

2. Distributed Bug Hunting

Of the many ways in which remote software monitoring can be used, our particular interest is in bug hunting tools. Industry critics have said that many software vendors treat their customers like beta testers. If that is so, then we are not yet using these thousands or millions of testers as effectively as we could. Traditionally, most software failures produce a grumpy user and no diagnostic feedback, which benefits no one. Recently, automatic crash reporting systems have created the reverse problem: developers who are overwhelmed with bug reports, many of which may be redundant, and who must prioritize their work in terms of which bug fixes are likely to provide the greatest net benefit in the shortest amount of time.

As of this writing, the Bugzilla bug tracking database for the open source Mozilla web browser project shows 36,937 open bugs; an additional 60,191 have been marked as duplicates of bugs already reported [6]. Mozilla augments manual bug reporting with an automated crash feedback system. This system currently shows 2974 automated crash reports over a ten day period, accounting for 12,799 hours of "testing" by end users [7]. Microsoft's Watson error reporting service has collected crash reports from half a million separate programs. Experience with Watson has shown that one percent of software errors cause fifty percent of user crashes [5].

This high level of redundancy suggests that there is great potential to harness the user community as a distributed, brute force bug hunting resource. Because the most important bugs are those that happen most often to the most users, it is not necessary to trace program behavior in a complete, invasive, perfectly controlled manner. Rather, we can use lightweight instrumentation to sample a small amount of information about each run, and then merge this information to form an aggregate picture of how the software is working and failing in the field. Furthermore, the feedback loop can flow in both directions: aggregate error reporting can direct engineers toward bugs, and engineers can steer instrumentation toward code regions of interest based on observed failure trends.

Any such system must solve several critical problems:

- If monitoring is to be continuous, any instrumentation must be sufficiently lightweight that it has a negligible

impact on the performance of the user's program.

- The data collection strategy must respect resource limits in several domains, including client (user) storage, server storage, and client-to-server network bandwidth.
- Given some instrumentation plan, analysis of collected data must draw probable conclusions from partial results, with progressively stronger inferences as more feedback data accumulates over time. We should not assume that we ever have complete information about any single run, or that any two runs are truly identical.

In the sections that follow, we describe ongoing research to address each of these areas. Section 3 describes a program transformation that produces a fair, randomized subset of some underlying instrumentation strategy. In Section 4 we couple sampled instrumentation with a statistical analysis based on logistic regression to isolate a buffer overrun bug.

3. Fair Random Sampling

Given some set of interesting program behaviors, it is impractical to observe all instances of these behaviors in all runs at all times. In one pilot study, we instrumented the `bc` command line calculator tool to trace the values of all syntactic assignments while processing nine megabytes of random input. An average run lasts less than five seconds (discounting instrumentation overhead) and yields over 250,000 assignment events. Using one word to identify the location of the assignment and a second word for its value, this data rate would require two full T1 Internet links (1.5 Mbps each) to stream just a single report to a feedback collection host. Even if traces were buffered locally, the overhead imposed by such aggressive instrumentation would impose an unacceptable performance penalty within the running client application.

Instead, we sample a sparse subset of behavior in a statistically fair, randomized manner. Given a body of code, with certain fragments designated as instrumentation, we wish to execute the instrumentation fragments only a subset of the times they are reached. With enough users, we can build up a realistic aggregate view while keeping the sampling density low enough that each individual user will experience only negligible instrumentation overhead. Furthermore, rather than reporting each sampled event individually, we fold the continuous event stream down to a finite collection of event counters within the client application. These counters represent a compact trace summary which can be reported and analyzed following program completion.

Our strategy is similar to one used by Arnold and Ryder for lightweight performance profiling [1], with refinements to ensure that the samples are truly random in the sense of a Bernoulli process: each instrumentation fragment has an equal chance of being executed or not, and this decision is made dynamically and independently at each instrumentation opportunity as the program executes.

Tossing a do/don't sample coin at each instrumentation site is too inefficient, since most instrumentation itself is already quite lightweight. Instead, at program compilation time, we identify acyclic regions in the control flow graph of each function. At the top of any acyclic region, there can be only a finite number of paths forward before we reach a back edge. Each such path can have only a finite number of instrumentation sites, and so the entire acyclic region has a finite maximum *instrumentation weight*.

At run time, we compute a sequence of random numbers in a geometric distribution. Geometrically distributed numbers give the *inter-arrival* time for events in a Bernoulli process; a geometric random sequence with mean 1000 tells us how many instrumentation sites to *skip* before taking the next sample for an overall sampling density of 1/1000.

Each geometrically distributed number, then, serves as a countdown relative to the instrumentation weights computed earlier. If the next sample countdown is 428, and we reach the top of an acyclic region with maximum instrumentation weight of 5, then no sample will be taken on this pass through the region. Instead of skipping over each instrumentation fragment individually, we jump directly into a "fast" version of the code in which the instrumentation has been removed entirely. All the fast code need do is decrement the countdown based on the actual path taken. When sampling is sparse, this is the common case, and so most execution will incur no instrumentation overhead beyond top-of-region counter checks and fast-path counter decrements.

Several opportunities exist for refining this sampling framework, including interprocedural analyses to identify larger acyclic regions and implementation strategies that help the compiler further optimize countdown management along the fast path. Additional details and preliminary experimental results have been published elsewhere [4].

4. Statistical Debugging

In this section we discuss one approach to using the information we collect to help isolate bugs that cause a program to crash. The framework described above limits per-client overhead, but captures only a sampled subset of behavior. Thus, while a crash is always observed, the violation that caused it may not be. Instead of looking for strict implications (assertion failed $\Longrightarrow$ crash), we are interested in finding statistical trends: if some assertion tends to hold when the program succeeds and tends to fail when the program fails, then this is an important clue to help a human programmer find, reproduce, and fix the bug.

Our approach injects instrumentation that guesses possible ordering relationships among pointer and integer variables in C programs, loosely in the style of Daikon [2]. At each direct scalar assignment "`x = ...`", we identify all same-typed variables $\{\mathtt{y}_1, \mathtt{y}_2, \ldots, \mathtt{y}_n\}$ which are simultaneously in scope. For each pair $(\mathtt{x}, \mathtt{y}_i)$, the compiler inserts a set of three comparisons to determine whether the new value assigned to `x` is less than, equal to, or greater than $\mathtt{y}_i$. Clearly these wild guesses will include many comparisons among variables which are completely unrelated, or which are always in some fixed relationship in all runs. However, it may also capture important relationships which directly relate to a bug. For example, an array bounds overrun appears as a pair of variables $(\mathtt{index}, \mathtt{max})$ for which `index` is always less than `max` on successful runs, but for which `index` is occasionally observed to be greater than `max` on failed runs.

Recording the result of each and every ordering comparison would yield a stream of samples that grows rapidly as the program executes and would exceed any reasonable constraints on storage or network bandwidth. Instead, the instrumentation maintains a triple of counters for each comparison site, and tabulates the number of times `x` was observed to be less than, equal to, or greater than $\mathtt{y}_i$. Each counter triple is considered as one sampling opportunity, and is randomly updated or skipped dynamically and independently from each other site. This can be thought of as a simple client-side analysis that applies summation as a reduction operator across the sample stream for each site and each ordering relation. When execution concludes, the counter values are shipped up to the feedback server along with a single binary outcome: failure or no failure. In a pilot study, full instrumentation of the `bc` command line calculator tool used 30,150 counters (thirty two bits each) of which just 2908 are "interesting" (non-zero in at least one run using 1/1000 sampling and 9MB random input). This much data easily fits within modest storage and communication resource limits.

Given hundreds or thousands of such runs, we borrow techniques from statistical analysis to identify those counters whose values are strongly predictive of program failure. Our current approach uses *logistic regression*, a discriminative binary classification method [3]. Whereas linear regression fits a straight line to input features, logistic regression uses an S-shaped curve (the logistic function) that asymptotically approaches zero at one extreme and one at the other. In our case, zero corresponds to successful execution, one corresponds to failure, input features are sampled counters, and the coefficient assigned to each feature weighs the relative importance of that feature in predicting program failure. The model is trained using stochastic gradient ascent to reach a local maximum of the log likelihood, with a penalty factor based on the L1-norm of the feature weight coefficients. We may have many more features than runs, and most features are wild guesses and therefore irrelevant. The penalty factor exerts a downward pressure forcing most feature weights to zero, leaving us with a small number of highly relevant predictors.

Our initial experience with the system is quite encouraging. We simulate a large user community by running `bc` with random input data. Nine megabytes of random input crashes `bc` 1.06 roughly one time in four. Stack traces show that `bc` failed in a utility routine under `malloc()`, suggesting heap corruption. We use 2704 runs with density 1/1000 to train the L1-regularized logistic regression model. The top five ranked features (those with the largest coefficients) form a group well-separated in magnitude from the rest. All five of these strongly predictive features correspond to instrumentation on a single line within a single function out of 8910 lines in the complete program. This line is the top of a loop, the counters correspond to updates of the loop index, and the function itself is clearly performing nontrivial memory management. Furthermore, these five features correspond to "greater-than" counters, suggesting that failure occurs when the loop index is unusually large. Inspection of the suspect line quickly reveals that the loop uses the wrong upper bound as it zeros out newly allocated elements of an array, causing it to run past the last element and scribble zeros into unmanaged memory.

One might hope that the specific invariant (that the index never exceed the array length) would appear as a unique strong predictor in the L1-regularized logistic regression model. This feature does appear in the model, but ranked 240th rather than first. There may be several reasons for this. Random noise is inherent to our methodology: both sampling and model training use randomization. Even crashing is not guaranteed: a C program may overrun some buffer, scribble into memory which is not actually being used, and therefore not crash. Lastly, there is a high degree of redundancy among instrumentation sites, meaning that the statistical model has several features to choose from which have equivalent predictive power. Improving our instrumentation scheme and fine-tuning the statistical analysis methodology are key areas for continued development.

The performance impact of sampled instrumentation is quite reasonable. For a sampling density of 1/1000, `bc` runs just 4.5% slower than it would with no instrumentation at all. By contrast, unconditional instrumentation without sampling incurs a penalty of over 390%. Thus, our sampling strategy allows large-scale distributed data collection that would be wholly impractical using naïve full tracing.

Further details about our bug hunting experiment with `bc` are reported elsewhere [4].

5. Open Problems

The techniques described here address challenges in building a distributed bug isolation system. However, a complete solution will require further study of additional open problems. To note just a few:

Bug report bucketing is the challenge of identifying failure reports which represent multiple occurrences of the same bug even before the bug itself has been found and fixed. A good bucketing scheme is key to assigning appropriate priorities, and is also a necessary basis for any approach that exploits redundancy to aggregate partial information from multiple failure reports. Clustering algorithms from machine learning may be applicable here.

Privacy and security are concerns with both social and technological dimensions. It may be possible to address some of the technological facets using techniques drawn from secure information flow, which statically identify sensitive data that should not be revealed to outside observers [8]. Our statistical models afford a degree of anonymity as well, as they combine many runs into one aggregate from which individual samples can no longer be recovered.

Adaptive refinement of instrumentation based on early results would allow us to focus in more quickly on targeted bugs while deemphasizing code that is known or believed to be uninvolved. Combining statistical models with more conventional program analysis techniques may allow us to make better initial instrumentation plans and refine these plans more rapidly than would either approach alone.

6. Conclusions

We have described a suite of instrumentation and analysis techniques for diagnosing bugs in widely deployed software. Bug isolation begins with continuous monitoring based on fair, sparsely sampled instrumentation. Statistical analysis based on L1-regularized logistic regression builds a predictive model that identifies sampled features which are highly predictive of subsequent program failure.

The strengths of the user community are twofold. First, they have overwhelming numbers. Second, they represent reality. All of our approaches are designed with the goal of leveraging these strengths. A very large user community means that random sampling can be sparse, which in turn allows us to control performance overhead. Simple reductions of the sampled data set on the client limit resource requirements, while still allowing statistical analysis of aggregate behavior. The logistic regression model grows progressively more accurate as it trains on more and more data, and will naturally adapt to reflect the most common failures. Redundant failure reports, therefore, are actually a benefit as they allow bug triage to reflect real failure trends seen by real users.

References

[1] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices*, 36(5):168–179, May 2001.

[2] 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, Feb. 2001.

[3] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning.* Stats. Springer, 2001.

[4] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Distributed program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[5] J. Markoff. Microsoft reports progress in averting computer crashes. *The New York Times*, page C.7, Oct. 3 2002.

[6] Mozilla.org. Mozilla bug database. `<http://bugzilla.mozilla.org/>`, Apr. 1 2003.

[7] Mozilla.org. Mozilla Talkback crash data. `<ftp://ftp.mozilla.org/pub/data/crash-data/>`, Apr. 1 2003.

[8] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 1–14. Chateau Lake Louise, Banff, Alberta, Canada, Oct. 2001. Appeared as ACM Operating Systems Review 35.5.