SCALABLE STATISTICAL BUG ISOLATION

Paper by Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, Michael I. Jordan

Presented by Ben Mishkanian

2/5/2015

Statistical Debugging

- Idea: Use dynamic statistical sampling to identify predicates that are highly correlated with program failure
- Example:

```
void buggyFunction(int x, int y) {
  int *nullPtr = null;
  if (x > 0) {
    nullPtr = alloc(int)
  }
  if (y < 5) {
    // insert irrelevant operation
  }
  *nullPtr = 3; // may crash
}
```

x > 0 perfectly predicts the crash

y < 5 predicts nothing, it's just a distraction

Motivation

- After deploying code, your users will likely encounter bugs your tests failed to find.
- A stack trace is of limited use. What you really want is the values of predicates that result in this bug.
- Full instrumentation is expensive.
- Statistical debugging correlates predicates with errors, but the performance is poor when there are multiple bugs.

Scalability Problems

- Redundant predictors
- Predicates predicting multiple bugs
- Bugs occur at different rates

Contributions of This Paper

- A scalable bug isolation algorithm
- A demonstration of effectiveness and efficiency
- An evaluation of the usefulness of stack traces
- A demonstration that this algorithm can find other types of failures, in addition to crashes

3 Examples of Predicates

Branch Predicates:

`if (x > 4)` has two predicates:

- branch taken
- branch not taken

Return Predicates:

`return returnVal` has six predicates:

- returnVal < 0
- returnVal <= 0
- returnVal > 0
- returnVal >= 0
- returnVal == 0
- returnVal != 0

3 Examples of Predicates

Scalar-Pairs Predicates:

Consider this snippet:

```
int CONST_VAL = 17;
bool b = True;
int x = 7;
int y = 8;
```

y has instrumentation sites comparing it to CONST_VAL and x, each containing 6 predicates, for a total of 12.

One predicate would be y > CONST_VAL

Multi-Bug Cause-Isolation Algorithm

1. Start with a set of runs and a set of known bugs.
2. Infer which predicates correspond to a single bug, and rank these predicates by importance to find a predictor.
3. Discard runs in which the predictor was ever true.
4. Pick a new bug and go to step #2.

Measuring Predicate Correlation

Estimate the probability of a crash given predicate P is true:

Let S(P) be # of successful runs where P is true

Let F(P) be # of failed runs where P is true

$$\text{Probability(Crash} \mid \text{P is true)} = \frac{F(P)}{S(P) + F(P)}$$

We call this value Failure(P)

False Predictors

- Problem: An irrelevant predicate can have high correlation with a crash, even if it doesn't cause the crash.
- Example:

```
if (f == NULL) {
  x = 0;
  *f;
}
```

Failure(P) is 1 at this predicate

But Failure(P) is also 1 at this irrelevant predicate

Use a Different Metric

- Instead of using Failure(P), we use Increase(P).

Let Context(P) = $\dfrac{\text{F(P observed)}}{\text{S(P observed) + F(P observed)}}$

Context(P) estimates the probability of failure when P is *seen*, rather than when P is true.

Increase(P) = Failure(P) – Context(P)

Increase(P) measures how much P being true *increases* the probability of failure.

False Predictor Avoided

```
if (f == NULL) {
  x = 0;
  *f;
}
```

Increase(P) is 1 here

Increase(P) is 0 here because Context(P) is already 1

Possible Ways To Rank Predicates

1. Sort by descending number of failures
   - Problem: predicates with many failures typically have weak correlations with particular bugs
2. Sort by Increase(P)
   - Problem: although these predicates have high correlations with bugs, they typically expose effects of a bug, rather than the root cause.

We need the sensitivity of #1, with the specificity of #2.

A Better Way

- The harmonic mean is a way to combine sensitivity and specificity.

$$Importance(P) = \frac{2}{\frac{1}{Increase(P)} + \frac{1}{log(F(P))/log(NumF)}}$$

- This formula balances the sensitivity of F(P) and the specificity of Increase(P)

The Precise Algorithm

1. Rank predicates by Importance
2. Remove the top-ranked predicate P and discard all runs in which P was ever true.
3. Repeat until either the set of runs or the set of predicates is empty.

Validation: Isolating Known Bugs

- Inserted nine bugs into the program MOSS
  - Six distinct bugs, plus three bugs that are variations of those
- Results:
  - Each of the top ranked predicates was a predictor for one specific bug
  - Most predicates had some runs triggering multiple bugs
  - A rare bug that was never triggered was not reported
- Note that we can also measure relationships between predicates P and P' by checking how removing runs correlated with P affects Importance(P').

Validation: RHYTHMBOX

- RHYTHMBOX is an event-driven system, so it is difficult to debug using static analysis or stack inspection.
- Results:
  - The top ranked predicate led to the discovery of a race condition
  - The second predicate had a correlated predicate that revealed a prevalent erroneous usage pattern of a library
  - Isolating the bugs took hours, but the predicates helped narrow down the problem areas

How Many Runs Are Needed?

- ~32,000 runs were used in the case studies, but this is overkill for getting a useful ranking of predictors.
- Testing EXIF, it was observed that 21,000 runs was sufficient to isolate the top three bugs, and only 2,000 runs to isolate the top two.
- Results degrade gracefully with fewer runs; predictors for rarer bugs are sacrificed first.

Comparison With Logistic Regression

- Logistic regression weights predicates to predict failure
- Flaw: it lumps all failures together
- Running logistic regression on MOSS, most of the predicates found predict sub-bugs or super-bugs

The Cooperative Bug Isolation Project

- Distributes instrumented versions of popular Linux software