Scalable Statistical
Bug Isolation

Ben Liblit, Mayur Naik, Alice Zheng,
Alex Aiken, and Michael Jordan
University of Wisconsin, Stanford
University, and UC Berkeley

Goal: Measure Reality

- Where is the black box for software?
  - Crash reporting systems are a start
- Actual runs are a vast resource
  - Number of real runs >> number of testing runs
  - Real-world executions are most important
- This talk: post-deployment bug hunting
  - Mining feedback data for causes of failure

What Should We Measure?

- Function return values
- Control flow decisions
- Minima & maxima
- Value relationships
- Pointer regions
- Reference counts
- Temporal relationships

In other words, lots of things

```
err = fetch(file, &obj);
if (!err && count < size)
  list[count++] = obj;
else
  unref(obj);
```

In other words, lots of things
Our Model of Behavior

Any interesting behavior is expressible as a predicate $P$ on program state at a particular program point.

Count how often “$P$ observed true” and “$P$ observed” using sparse but fair random samples of complete behavior.

Bug Isolation Architecture

Find Causes of Bugs

- Gather information about many predicates
  - 298,482 predicates in bc
  - 857,384 predicates in Rhythmbox
- Most are not predictive of anything
- How do we find the useful bug predictors?
  - Data is incomplete, noisy, irreproducible, …

Look For Statistical Trends

How likely is failure when $P$ happens?

$$F(P) = \text{# of failures where } P \text{ observed true}$$

$$S(P) = \text{# of successes where } P \text{ observed true}$$

$$\text{Failure}(P) = \frac{F(P)}{F(P) + S(P)}$$
if (f == NULL) {  
   x = 0;
   *f;
}

• Predicate x == 0 is an innocent bystander
  – Program is already doomed

Good Start, But Not Enough

Context

What is the background chance of failure regardless of P’s truth or falsehood?

$F(P \text{ observed}) = \# \text{ of failures observing } P$

$S(P \text{ observed}) = \# \text{ of successes observing } P$

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

Isolate the Predictive Value of P

Does P being true increase the chance of failure over the background rate?

$\text{Increase}(P) = \text{Failure}(P) - \text{Context}(P)$

(a form of likelihood ratio testing)

Increase() Isolates the Predictor

if (f == NULL) {  
   x = 0;
   *f;
}
void more_arrays ()
{
  ...
  /* Copy the old arrays. */
  for (indx = 1; indx < old_count; indx++)
    arrays[indx] = old_ary[indx];
  /* Initialize the new elements. */
  for (; indx < v_count; indx++)
    arrays[indx] = NULL;
  ...
}

Isolating a Single Bug in bc

#1: indx > scale
#2: indx > use_math
#3: indx > opterr
#4: indx > next_func
#5: indx > i_base

It Works!

...for programs with just one bug.

- Need to deal with multiple, unknown bugs
- Redundant predictors are a major problem

Goal: Isolate the best predictor for each bug, with no prior knowledge of the number of bugs.

Multiple Bugs: Some Issues

- A bug may have many redundant predictors
  - Only need one, provided it is a good one
- Bugs occur on vastly different scales
  - Predictors for common bugs may dominate, hiding predictors of less common problems

Guide to Visualization

- Multiple interesting & useful predicate metrics
- Graphical representation helps reveal trends

Increase(P)  error bound
Context(P)

log(F(P) + S(P))

S(P)
Bad Idea #1: Rank by $\text{Increase}(P)$

- High $\text{Increase}()$ but very few failing runs!
- These are all sub-bug predictors
  - Each covers one special case of a larger bug
- Redundancy is clearly a problem

Bad Idea #2: Rank by $\text{F}(P)$

- Many failing runs but low $\text{Increase}()$
- Tend to be super-bug predictors
  - Each covers several bugs, plus lots of junk

A Helpful Analogy

- In the language of information retrieval
  - $\text{Increase}(P)$ has high precision, low recall
  - $\text{F}(P)$ has high recall, low precision
- Standard solution:
  - Take the harmonic mean of both
  - Rewards high scores in both dimensions

Rank by Harmonic Mean

- It works!
  - Large increase, many failures, few or no successes
- But redundancy is still a problem
Redundancy Elimination

- One predictor for a bug is interesting
  - Additional predictors are a distraction
  - Want to explain each failure once

- Similar to minimum set-cover problem
  - Cover all failed runs with subset of predicates
  - Greedy selection using harmonic ranking

Simulated Iterative Bug Fixing

1. Rank all predicates under consideration
2. Select the top-ranked predicate $P$
3. Add $P$ to bug predictor list
4. Discard $P$ and all runs where $P$ was true
   - Simulates fixing the bug predicted by $P$
   - Reduces rank of similar predicates
5. Repeat until out of failures or predicates

Experimental Results: `exif`

- 3 bug predictors from 156,476 initial predicates
- Each predicate identifies a distinct crashing bug
- All bugs found quickly using analysis results
Experimental Results: Rhythmbox

- 15 bug predictors from 857,384 initial predicates
- Found and fixed several crashing bugs

Lessons Learned

- Can learn a lot from actual executions
  - Users are running buggy code anyway
  - We should capture some of that information
- Crash reporting is a good start, but…
  - Pre-crash behavior can be important
  - Successful runs reveal correct behavior
  - Stack alone is not enough for 50% of bugs

Public Deployment in Progress

Join the Cause!

The Cooperative Bug Isolation Project

http://www.cs.wisc.edu/cbi/
Effectively Prioritizing Tests in Development Environment

Amitabh Srivastava
Jay Thiagarajan
PPRC, Microsoft Research

Using program changes

- Source code differencing
  - F. Vokolos & P. Frankl, “Pythia: a regression test selection tool based on text differencing”, May 1997

- Data and control flow analysis

- Code entities

Analysis of various techniques

- Source code differencing
  - Simple and fast
  - Can be built using commonly available tools like “diff”
  - Simple renaming of variable will trip off
  - Will fail when macro definition changes
  - To avoid these pitfalls, static analysis is needed

- Data and control flow analysis
  - Flow analysis is difficult in languages like C/C++ with pointers, casts and aliasing
  - Interprocedural data flow techniques are extremely expensive and difficult to implement in complex environment
Our Solution

- Focus on change from previous version
  - Determine change at very fine granularity – basic block/instruction

- Operates on binary code
  - Easier to integrate in production environment
  - Scales well to compute results in minutes

- Simple heuristic algorithm to predict which part of code is impacted by the change

Test Effectiveness Infrastructure

- Repository
- Coverage Tools
  - Test Prioritization
  - Magellan

Coverage Impact Analysis

- Old Build
- New Build
  - Binary Diff
  - BMAT/VULCAN
  - Test Prioritization

Echelon : Test Prioritization

- Repository
- Magellan
  (*link with symbol server for symbols)

Leverage what has already been tested

Prioritized list of test cases

Block Change Analysis: Binary Matching

- Old Build
- New Build
  - Old Blocks (not changed)
  - Old Blocks (changed)
  - New Blocks

BMAT – Binary Matching [Wang, Pierce and McFarling JILP 2000]
**Coverage Impact Analysis**

- **Terminology**
  - Trace: collection of one or more test cases
  - Impacted Blocks: old modified and new blocks

- Compute the coverage of traces for the new build
  - Coverage for old (unchanged and modified) blocks are same as the coverage for the old build
  - Coverage for new nodes requires more analysis

- **Limitations - New node may not be executed**
  - If there is a path from successor to predecessor
  - If there are changes in control path due to data changes

---

**Coverage Impact Analysis**

- **A Trace may cover a new block N if it covers at least one Predecessor block and at least one Successor Block**

- **If P or S is a new block, then its Predecessors or successors are used (iterative process)**

---

**Echelon: Test Case Prioritization**

- Detects minimal sets of test cases that are likely to cover the impacted blocks (old changed and new blocks)
  - Input is traces (test cases) and a set of impacted blocks
  - Uses a greedy iterative algorithm for test selection
Echelon: Test Selection

Set 1
- T1
- T2

Set 2
- T3
- T5

Set 3
- T4

Weights:

\[ T1 \]
\[ T2 \]
\[ T3 \]
\[ T4 \]
\[ T5 \]

Denotes that a trace T covers the impacted block

Echelon: Test Selection Output

Ordered List of Traces

Set 1
- T1
- T2

Set 2
- T3
- T5

Set 3
- T4
- T7
- T8...
- Tm

Each set contains test cases that will give maximum coverage of impacted nodes.

Gracefully handles the "main" modification case.

If all the test can be run, tests should be run in the order to maximize the chances of detecting failures early.

Analysis of results

Three measurements of interest
- How many sequences of tests were formed?
- How effective is the algorithm in practice?
- How accurate is the algorithm in practice?

Details of BinaryE

<table>
<thead>
<tr>
<th></th>
<th>Version 1</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>12/11/2000</td>
<td>01/29/2001</td>
</tr>
<tr>
<td>Functions</td>
<td>31,020</td>
<td>31,026</td>
</tr>
<tr>
<td>Blocks</td>
<td>668,068</td>
<td>668,274</td>
</tr>
<tr>
<td>Arcs</td>
<td>1,097,294</td>
<td>1,097,650</td>
</tr>
<tr>
<td>File size</td>
<td>8,880,128</td>
<td>8,880,128</td>
</tr>
<tr>
<td>PDB size</td>
<td>22,602,752</td>
<td>22,651,904</td>
</tr>
<tr>
<td>Impacted Blocks</td>
<td>0</td>
<td>378 (220 N, 158 OC)</td>
</tr>
<tr>
<td>Number of Traces</td>
<td>3128</td>
<td>3128</td>
</tr>
<tr>
<td># Source Lines</td>
<td>~1.8 Million</td>
<td>~1.8 Million</td>
</tr>
</tbody>
</table>

Echelon takes ~210 seconds for this 8MB binary.
Effectiveness of Echelon

- Important Measure of effectiveness is early defect detection
- Measured % of defects vs. % of unique defects in each sequence
- Unique defects are defects not detected by the previous sequence
Effectiveness of Echelon

Defects detected in each sequence

% Defects detected

% Unique Defects

Sequence

Defects detected in each sequence

% Defects detected

% Unique Defects

Sequence

Blocks predicted hit that were not hit

Blocks predicted not hit that were actually hit

(Blocks were target of indirect calls are being predicted as not hit)
Summary

- Binary based test prioritization approach can effectively prioritize tests in large scale development environment
- Simple heuristic with program change in fine granularity works well in practice
- Currently integrated into Microsoft Development process

Coverage Impact Analysis

- Echelon provides a number of options
  - Control branch prediction
  - Indirect calls: if N is target of an indirect call a trace needs to cover at least one of its successor block
- Future improvements include heuristic branch prediction
  - Branch Prediction for Free [Ball, Larus]
Echelon: Test Selection

- Options
- Calculations of weights can be extended, e.g. traces with great historical fault detection can be given additional weights
- Include time each test takes into calculation
- Print changed (modified or new) source code that may not be covered by any trace
- Print all source code lines that may not be covered by any trace