

Cooperative Bug Isolation

Alex Aiken

Mayur Naik

Stanford University



Stanford University



Ben Liblit

University of Wisconsin

Alice Zheng

Michael Jordan

UC Berkeley



UC Berkeley

Build and Monitor



Alex Aiken, Cooperative Bug Isolation

The Goal: Analyze Reality



- Where is the black box for software?
 - Historically, some efforts, but spotty
 - Now it's really happening: crash reporting systems
- Actual runs are a vast resource
 - Number of real runs \gg number of testing runs
 - And the real-world executions are most important
- This talk: post-deployment bug hunting

Engineering Constraints



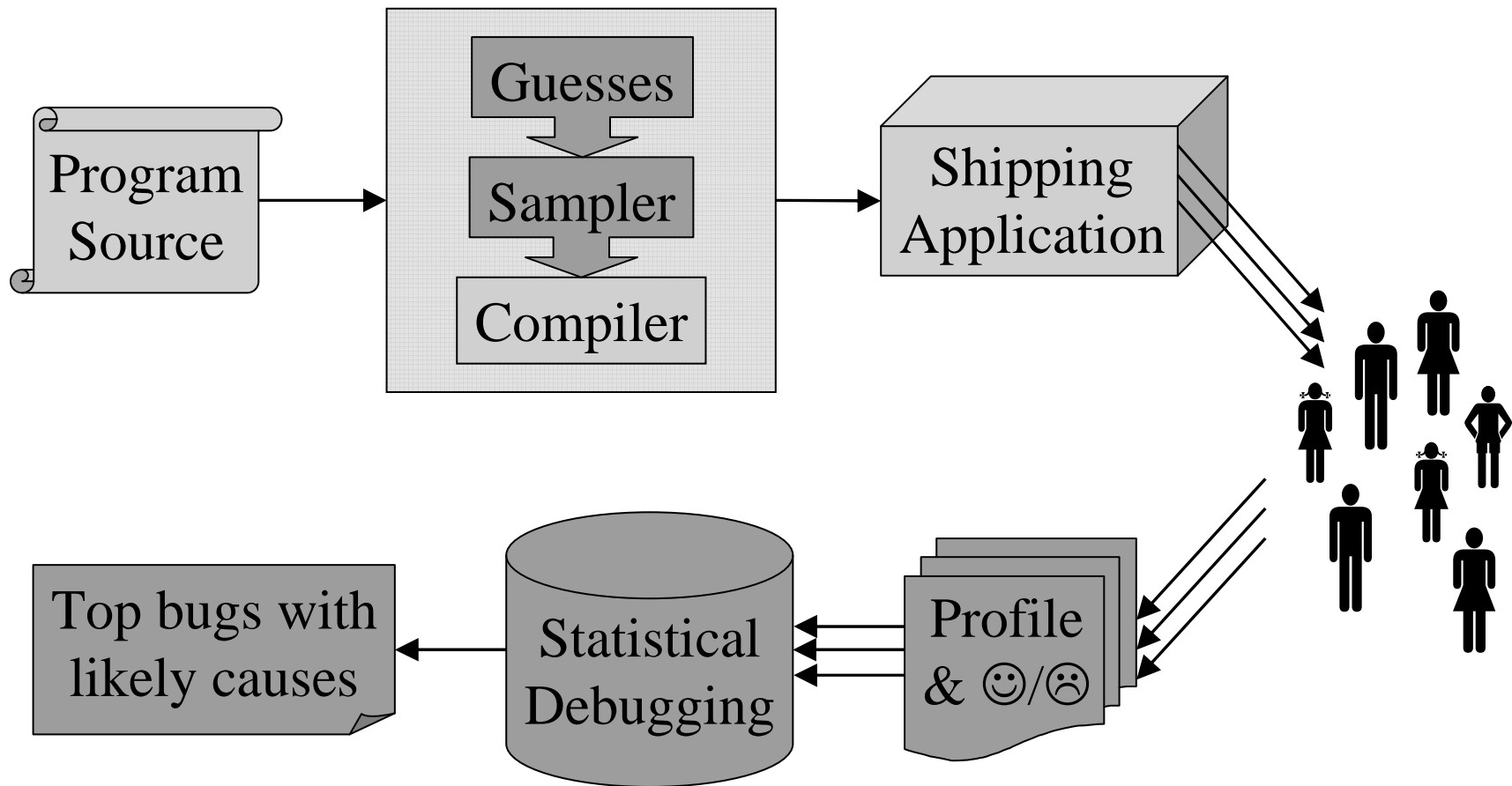
- Big systems
 - Millions of lines of code
 - Mix of controlled, uncontrolled code
 - Threads
- Remote monitoring
 - Limited disk & network bandwidth
- Incomplete information
 - Limit performance overhead
 - Privacy and security

The Approach



1. Guess "potentially interesting" behaviors
 - Compile-time instrumentation
2. Collect sparse, fair subset of these behaviors
 - Generic sampling transformation
 - Feedback profile + outcome label
3. Find behavioral changes in good/bad runs
 - Statistical debugging

Bug Isolation Architecture



Our Model of Behavior



We assume any interesting behavior is expressible as a predicate P on program state at a particular program point.

Observation of behavior = observing P

Branches Are Interesting



```
if (p) ...  
else  ...
```



Branch Predicate Counts

```
++branch_17[!!p];  
if (p) ...  
else ...
```

- Predicates are folded down into counts
- C idiom: `!!p` ensures subscript is 0 or 1

Return Values Are Interesting



```
n = fprintf(...);
```



Returned Value Predicate Counts

```
n = fprintf(...);  
++call_41[ (n==0) + (n>=0) ] ;
```

- Track predicates: $n < 0$, $n == 0$, $n > 0$

Scalar Relationships



```
int i, j, k;
```

```
...
```

```
i = ...i
```

The relationship of `i` to other integer-valued variables in scope after the assignment is potentially interesting...



Pair Relationship Predicate Counts

```
int i, j, k;
```

```
...
```

```
i = ...i
```

Is $i < j$, $i = j$, or $i > j$?

```
++pair_6 [ (i==j) + (i>=j) ] ;
```

```
++pair_7 [ (i==k) + (i>=k) ] ;
```

```
++pair_8 [ (i==5) + (i>=5) ] ;
```

Test i against all other constants & variables in scope.

Summarization and Reporting



- Instrument the program with predicates
 - We have a variety of instrumentation schemes

- Feedback report is:
 - Vector of predicate counters
 - Success/failure outcome label

P1	P2	P3	P4	P5	...
0	0	4	0	1	...

- No time dimension, for good or ill
- Still quite a lot to measure
 - What about performance?

Sampling



- Decide to examine or ignore each site...
 - Randomly
 - Independently
 - Dynamically
- Why?
 - Fairness
 - We need accurate picture of rare events.



Problematic Approaches

- Sample every k th predicate
 - Violates independence
- Use clock interrupt
 - Not enough context
 - Not very portable
- Toss a coin at each instrumentation site
 - Too slow



Amortized Coin Tossing

- Observation
 - Samples are rare, say $1/100$
 - Amortize cost by predicting time until next sample
- Randomized global countdown
 - Small countdown \Rightarrow upcoming sample
- Selected from *geometric distribution*
 - Inter-arrival time for biased coin toss
 - How many tails before next head?

Geometric Distribution



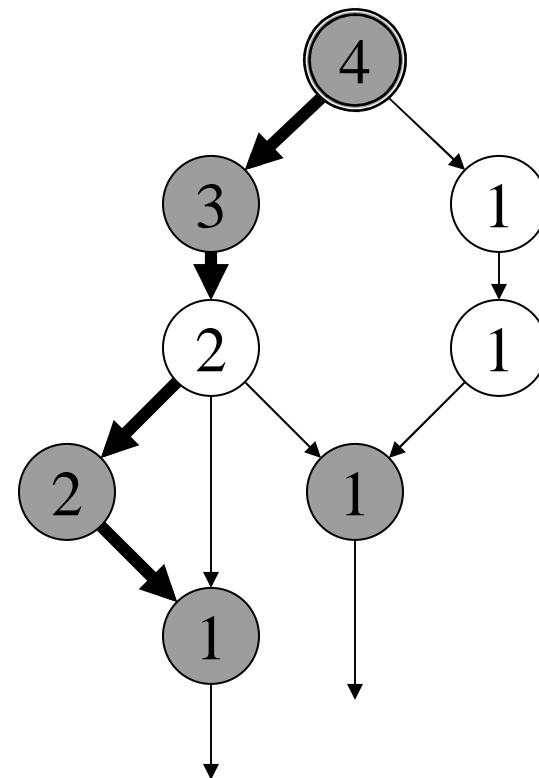
$$next = \left\lfloor \frac{\log(rand(0,1))}{\log(1 - 1/D)} \right\rfloor + 1$$

D = mean of distribution
= expected sample density



Weighing Acyclic Regions

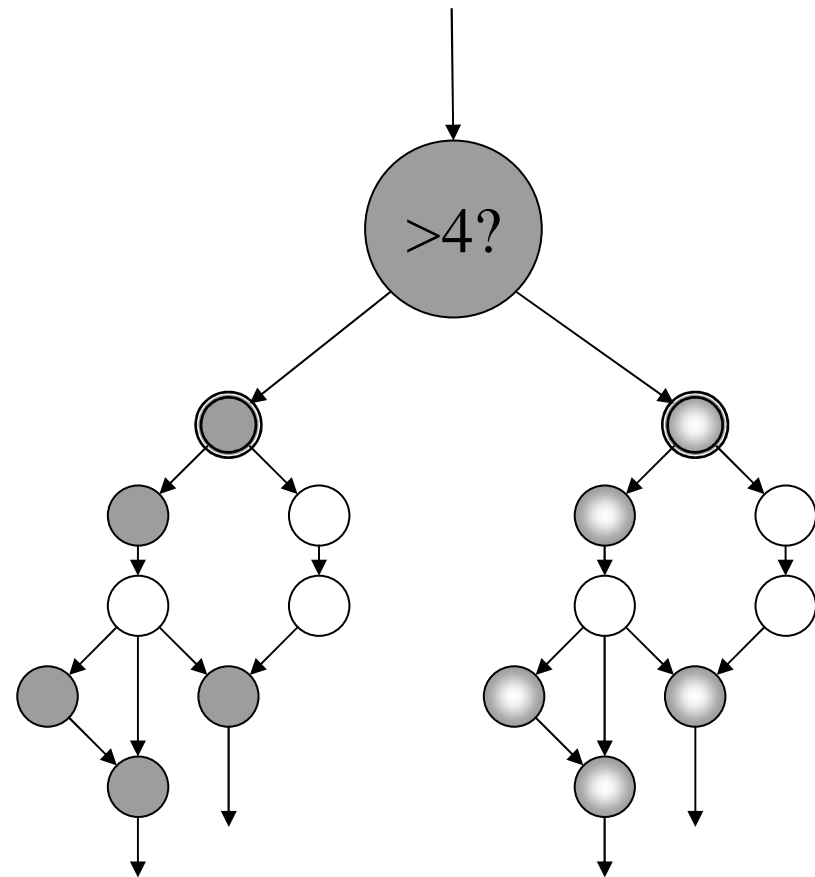
- An acyclic region has:
- a finite number of paths
- a finite max number of instrumentation sites executed





Weighing Acyclic Regions

- Clone acyclic regions
 - "Fast" variant
 - "Slow" variant
- Choose at run time based on countdown to next sample

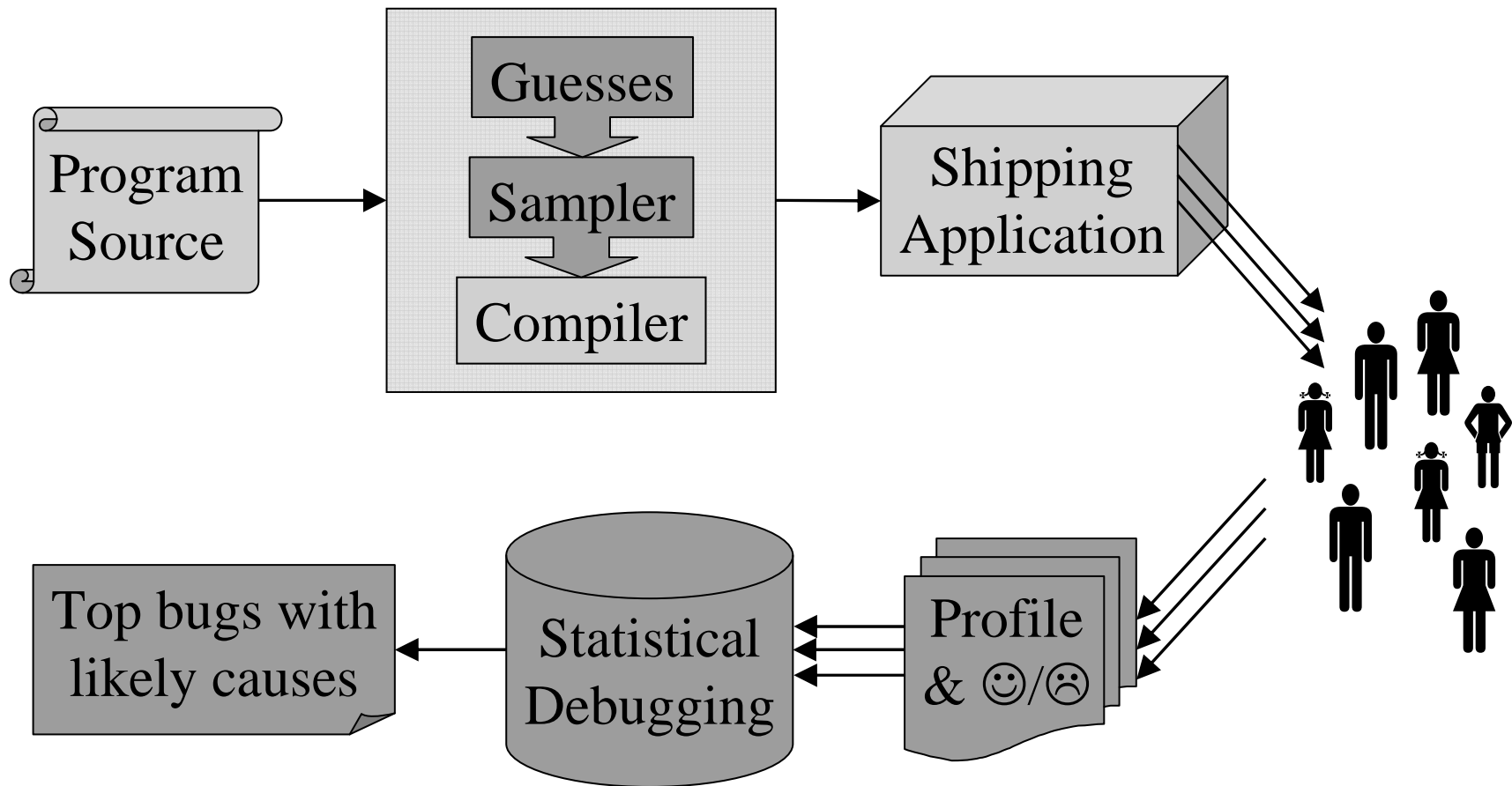


Summary: Feedback Reports



- Subset of dynamic behavior
 - Counts of true/false predicate observations
 - Sampling gives low overhead
 - Often unmeasurable at 1/100
 - Success/failure label for entire run
- Certain of what we did observe
 - But may have missed some events
- Given enough runs, samples \approx reality
 - Common events seen most often
 - Rare events seen at proportionate rate

Bug Isolation Architecture



Find Causes of Bugs



- We gather information about *many* predicates.
 - 298,482 for BC
- Most of these are not predictive of anything.
- How do we find the useful predicates?

Finding Causes of Bugs



How likely is failure when P is observed true?

$F(P)$ = # failing runs where P observed true

$S(P)$ = # successful runs where P observed true

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

Not Enough . . .



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

Failure(f == NULL) = 1.0

Failure(x == 0) = 1.0

- Predicate $x == 0$ is an innocent bystander
 - Program is already doomed

Context



What is the background chance of failure, regardless of P's value?

$F(P \text{ observed}) = \# \text{ failing runs observing } P$

$S(P \text{ observed}) = \# \text{ successful runs observing } P$

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

A Useful Measure



Does the predicate 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() Works . . .



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

Increase(f == NULL) = 1.0

Increase(x == 0) = 0.0

A First Algorithm



1. Discard predicates having $\text{Increase}(P) \leq 0$
 - E.g. dead, invariant, bystander predicates
 - Exact value is sensitive to small $F(P)$
 - Use lower bound of 95% confidence interval

2. Sort remaining predicates by $\text{Increase}(P)$
 - Again, use 95% lower bound
 - Likely causes with determinacy metrics

Isolating a Single Bug in BC



```
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;

  ...
}
```

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

It Works!

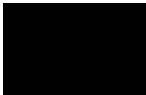


- Well . . . at least for a program with 1 bug.
- But
 - Need to deal with multiple, unknown bugs.
 - Redundancy in the predicate list is a major problem.

Using the Information



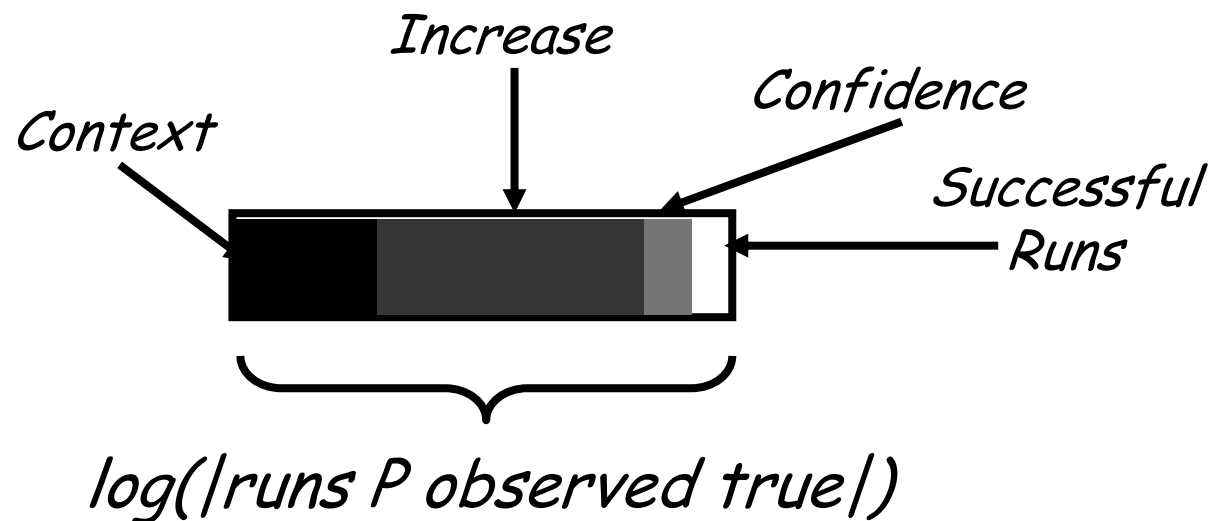
- Multiple predicate metrics are useful
 - Increase(P), Failure(P), F(P), S(P)

 ext(P) = .25



$F(P) + S(P) = 349$

The Bug Thermometer



Sample Report



http://xenon.stanford.edu/~mhn/B_lb.html - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://xenon.stanford.edu/~mhn/B_lb.html

Google alex aiken

Scheme: [branch] [return] [scalar] [all]

Sort by: [lower bound of confidence interval] [increase score] [fail score] [true in # F runs]

thermometer	predicate	function	file:line
	tmp__5 is FALSE	info_available_cb	rb-shell-player.c:1774
	tmp__6 is FALSE	rb_shell_jump_to_entry_with_source	rb-shell.c:2118
	tmp is TRUE	rhythmdb_tree_entry_insert	rhythmdb-tree.c:838
	(mp->priv)->timer is FALSE	monkey_media_player_finalize	monkey-media-player-gst-tmp.c:241
	(rorder->priv)->source is FALSE	rb_random_play_order_by_age_finalize	rb-play-order-random-by-age.c:184
	(hist->priv)->db is FALSE	rb_history_finalize	rb-history.c:190
	(unsigned int)ptr == (unsigned int)((void *)0) is TRUE	rhythmdb_query_model_entry_to_iter	rhythmdb-query-model.c:872
	tmp__8 is FALSE	remove_entry_from_album	rhythmdb-tree.c:1030
	tmp is TRUE	eel_gconf_get_boolean	eel-gconf-extensions.c:107
	(unsigned int)(*error) != (unsigned int)((void *)0) is TRUE	eel_gconf_handle_error	eel-gconf-extensions.c:66
	(db->priv)->outstanding_threads > 0 is TRUE	rhythmdb_shutdown	rhythmdb.c:369
	(statusbar->priv)->idle_tick_id is TRUE	rb_statusbar_finalize	rb-statusbar.c:280
	! ((int)(db->priv)->changed_entries) is FALSE	rhythmdb_entry_set	rhythmdb.c:1001
	(view->priv)->change_sig_queued is TRUE	rb_entry_view_finalize	rb-entry-view.c:416

Multiple Bugs: The 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
 - But would like to know correlated predictors
- Bugs occur on vastly different scales
 - Predictors for common bugs may dominate, hiding predictors of less common problems



An Idea

- Simulate the way humans fix bugs
- Find the first (most important) bug
- Fix it, and repeat

An Algorithm



Repeat the following:

1. Compute `Increase()`, `Context()`, etc. for all preds.
2. Rank the predicates
3. Add the top-ranked predicate P to the result list
4. Remove P & discard all runs where P is true
 - Simulates fixing the bug corresponding to P
 - Discard reduces rank of correlated predicates

Bad Idea #1: Ranking by Increase(P)



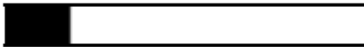

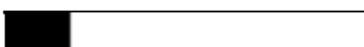


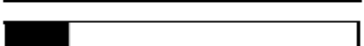
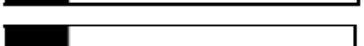
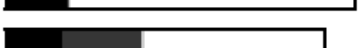
Thermometer	Context	Increase	S	F
██████████	0.065	0.935 ± 0.019	0	23
██████	0.065	0.935 ± 0.020	0	10
██████████	0.071	0.929 ± 0.020	0	18
██████	0.073	0.927 ± 0.020	0	10
██████████	0.071	0.929 ± 0.028	0	19
██████	0.075	0.925 ± 0.022	0	14
██████	0.076	0.924 ± 0.022	0	12
██████	0.077	0.923 ± 0.023	0	10

High Increase() but very few failing runs!

These are all *sub-bug predictors*: they cover a special case of a more general problem.

Bad Idea #2: Ranking by Fail(P)



Thermometer	Context	Increase	S	F
	0.176	0.007 ± 0.012	22554	5045
	0.176	0.007 ± 0.012	22566	5045
	0.176	0.007 ± 0.012	22571	5045
	0.176	0.007 ± 0.013	18894	4251
	0.176	0.007 ± 0.013	18885	4240
	0.176	0.008 ± 0.013	17757	4007
	0.177	0.008 ± 0.014	16453	3731
	0.176	0.261 ± 0.023	4800	3716

Many failing runs but low Increase()!

Tend to be *super-bug predictors*: predicates that cover several different bugs rather poorly.



A Helpful Analogy

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

Ranking by the Harmonic Mean



Thermometer	Context	Increase	S	F
	0.176	0.824 ± 0.009	0	1585
	0.176	0.824 ± 0.009	0	1584
	0.176	0.824 ± 0.009	0	1580
	0.176	0.824 ± 0.009	0	1577
	0.176	0.824 ± 0.009	0	1576
	0.176	0.824 ± 0.009	0	1573
	0.116	0.883 ± 0.012	1	774
	0.116	0.883 ± 0.012	1	776

It works!

Experimental Results: `Exif`



Initial	Effective	Predicate
		<code>i < 0</code>
		<code>maxlen > 1900</code>
		<code>o + s > buf size is TRUE</code>

- Three predicates selected from 156,476
- Each predicate predicts a distinct crashing bug
- We found the bugs quickly using these predicates

Experimental Results: Rhythmbox

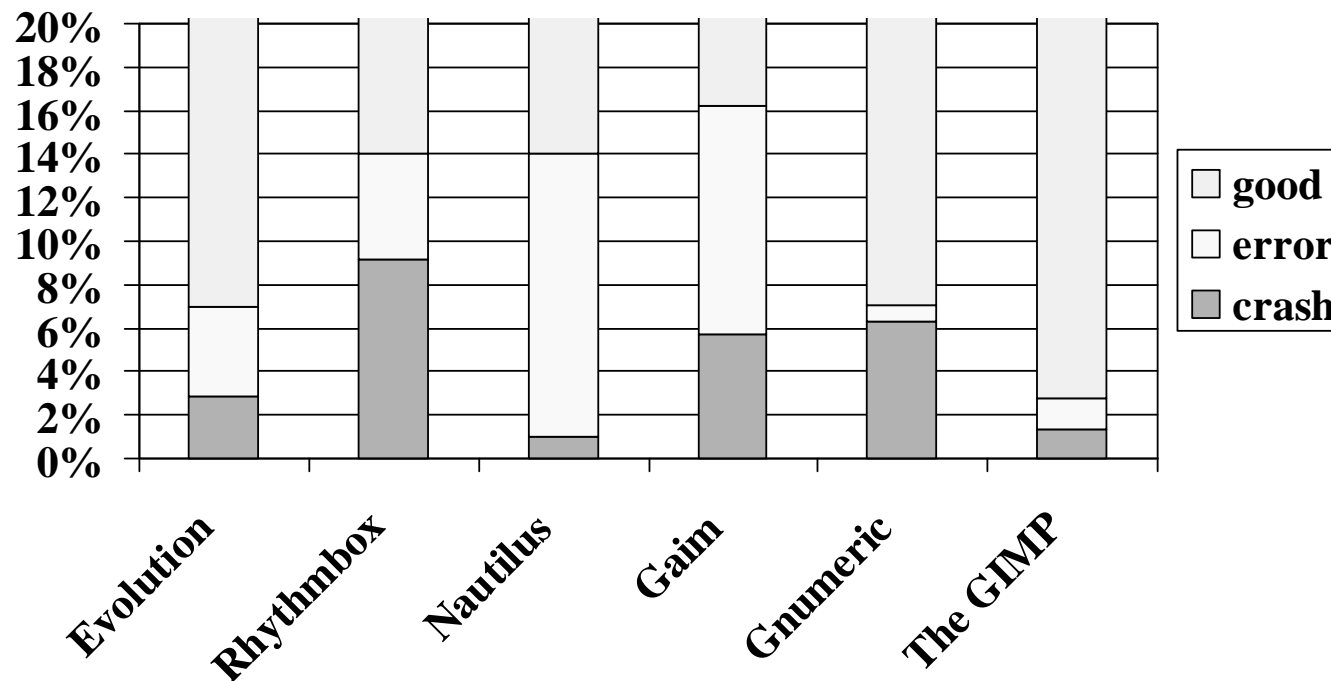


Initial	Effective	Predicate
		<code>tmp is FALSE</code>
		<code>(mp->priv)->timer is FALSE</code>
		<code>(view->priv)->change_sig_queued is TRUE</code>
		<code>(hist->priv)->db is TRUE</code>
		<code>rb_playlist_manager_signals[0] > 269</code>
		<code>(db->priv)->thread_reaper_id >= 12</code>
		<code>entry == entry</code>
		<code>fn == fn</code>
		<code>klass > klass</code>
		<code>genre < artist</code>
		<code>vol <= (float)0 is TRUE</code>
		<code>(player->priv)->handling_error is TRUE</code>
		<code>(statusbar->priv)->library_busy is TRUE</code>
		<code>shell < shell</code>
		<code>len < 270</code>

- 15 predicates from 857,384

- Also isolated crashing bugs . . .

Public Deployment in Progress



Lessons Learned



- A lot can be learned from actual executions
 - Users are executing them anyway
 - We should capture some of that information
- Crash reporting is a step in the right direction
 - But doesn't characterize successful runs
 - Stack is useful for only about 50% of bugs
- Bug finding is just one possible application
 - Understanding usage patterns
 - Understanding performance in different environments



Related Work

- *Gamma*
 - Georgia Tech
- *Daikon*
 - MIT/U. Washington
- *Diduce*
 - Stanford

The Cooperative Bug Isolation Project

<http://www.cs.wisc.edu/cbi/>

