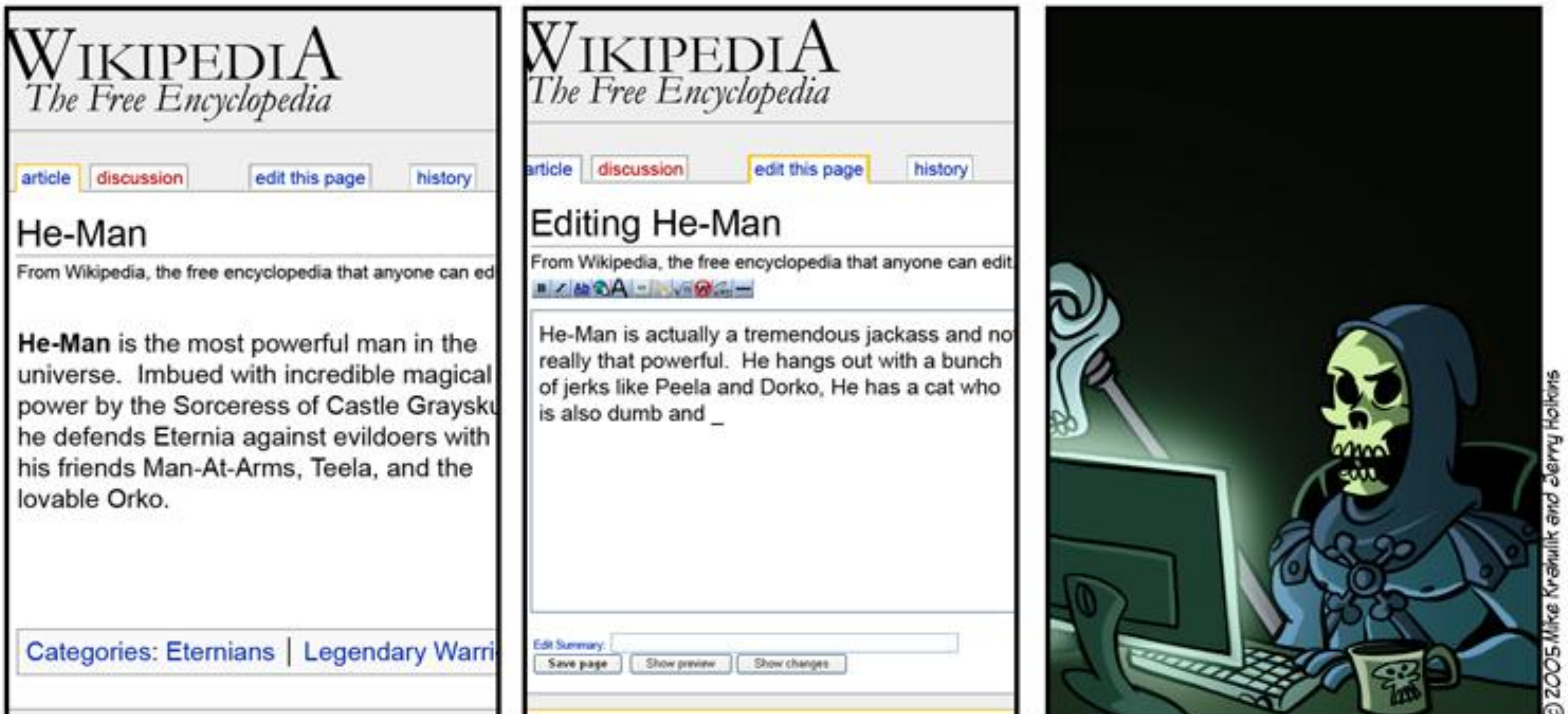
Cooperative Bug Isolation

Ben Liblit *et al.*



What’s This?

Today, we'll talk about the work that won the 2005 ACM Doctoral Dissertation Award.

Take Home Message

- Bugs experienced by users matter.
- We can use information from user runs of programs to find bugs.
- Random sampling keeps the overhead of doing this low.
- Large public deployments exist.

Reading Quiz

Why this work is cool

- Crosscutting insights
  - PL, SE, ML, Stats, …
- Simple idea, challenging and long-running research project.
  - Has spawned at least 17 papers
- Real world impact
  - "Thanks to Ben Liblit and the Cooperative Bug Isolation Project, this version of Rhythmbox should be the most stable yet."

Today's Goal: Measure Reality

We measure bridges, airplanes, cars…

- Where is the right data recorder for software?

Today’s Goal: Measure Reality

Users are a vast, untapped resource

- 60 million XP licenses in first year; 2/second
- 1.9M Kazaa downloads per week in 2004; 3/s
- Users know what matters most
  - Nay, users *define* what matters most!

Opportunity for *reality-directed* debugging

- Implicit bug triage for an imperfect world

Good News Everyone!

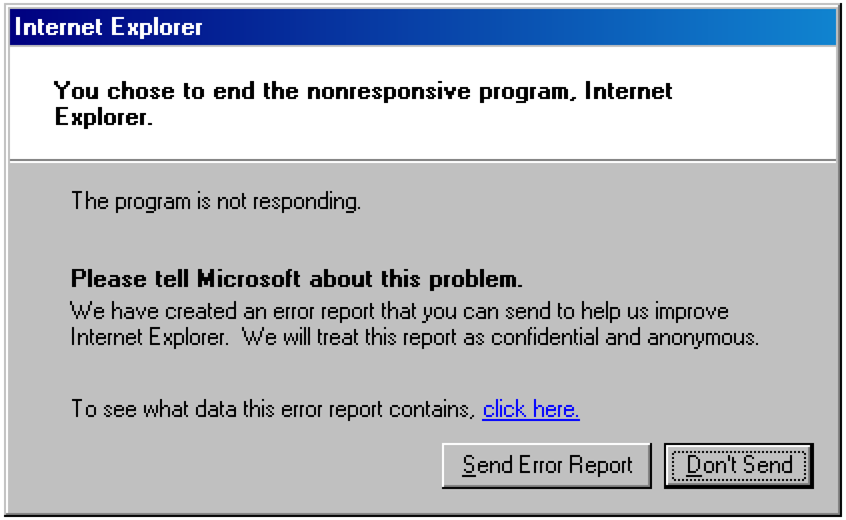
- Users can help!
- Important bugs happen often, to many users
  - User communities are big and growing fast
  - **User runs vastly exceed testing runs**
  - Users are networked

“There are no significant bugs in our released software that any significant number of users want fixed.”

-- Bill Gates in 1995

Possibility 1: Crash Reports

- In use since mid 90s
- Stack trace, memory addresses, details about host configuration, …
- Advantage: fast + easy
- Limitations:
  - Crashes don’t always occur “near” the defect
  - Hard to tell which crashes correspond to the same bug



Possibility 2:
Instrument all the things

- Advantage:
  - You have everything you could hope to have for bug triage
- Limitation:
  - Way to slow for deployed code
    - Heavyweight instrumentation like this implies ~1000x slowdown

The best of both worlds

- Lots of information: something close to what a debugger could tell us
- Ability to compare failed runs to good runs
- No compromise on performance for users

Solution: Randomness

Similar to statistical profiling

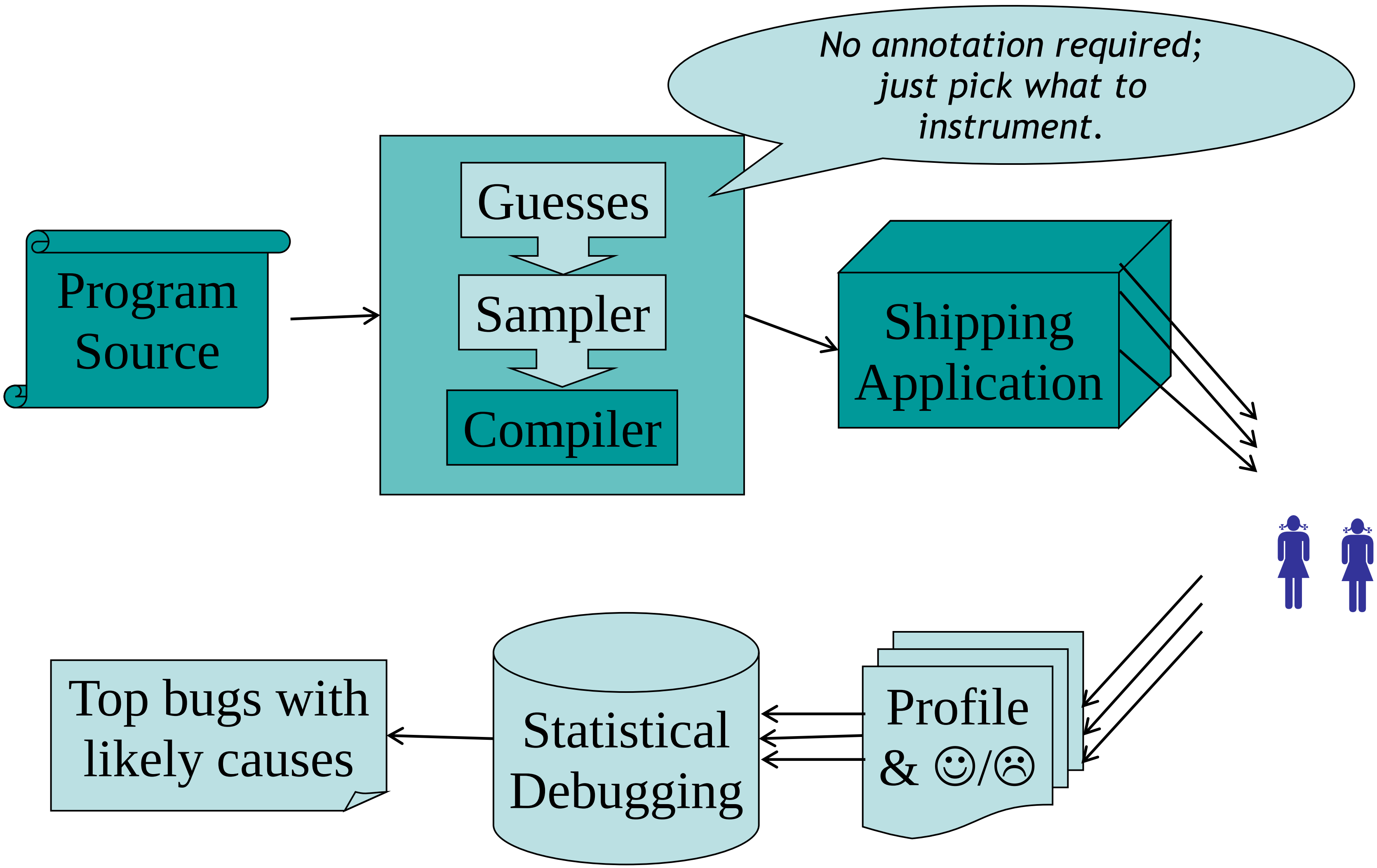
- AMD CodeAnalyst, Shark, gprof, Intel VTune

Idea: each user records 0.1% of everything

Generic sparse sampling framework

- Adaptation of Arnold & Ryder

Bug Isolation Architecture



Potential Concerns

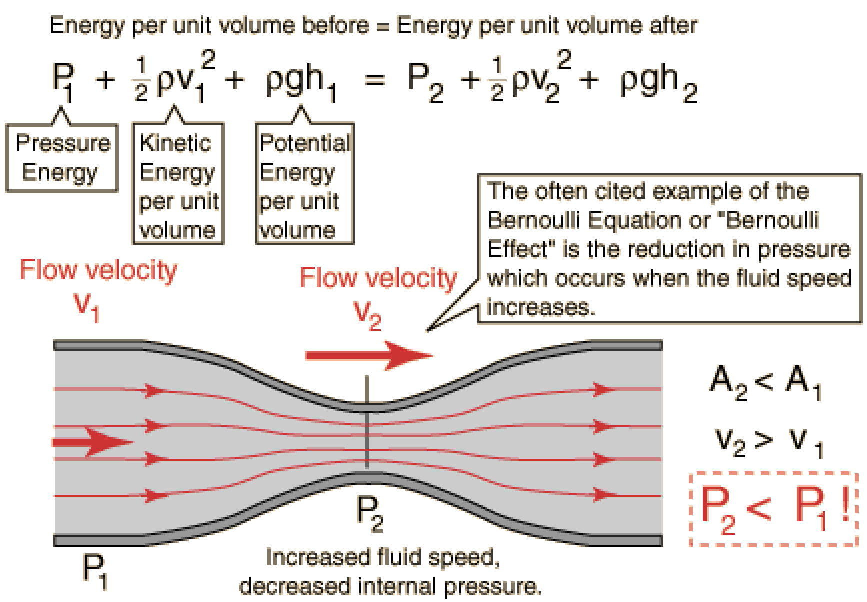
- Flipping a random coin isn't cheap
- Many kind of things we might like to record
  - Function return values, Control flow decisions, Minima & maxima, Value relationships, Pointer regions, Reference counts, Temporal relationships
- Aggregated data may be huge
  - $\Rightarrow$ Client-side reduction/summarization
- Will never have complete information
  - $\Rightarrow$ Make wild guesses about bad behavior
  - $\Rightarrow$ Look for broad trends across many runs

Sampling

Identify the points of interest

Decide to examine or ignore each site…

- Randomly
- Independently
- Dynamically



Sampling Blocks

Consider the following piece of code

```
check(p != NULL);
p = p->next;
check(i < max);
total += sizes[i];
```

We want to sample 1/100th of these checks

Sampling Blocks

Solution 1 : Maintain a global counter modulo 100

Problem?

```
for(i = 0 ; i < n; i++)
 {
  check(p != NULL);
  p = p->next;
  check(i < max);
  total += sizes[i];
 }
```

Sampling Blocks

Solution 1 : Maintain a global counter modulo 100

Problem?

```
for(i = 0 ; i < n; i++)
 {
  check(p != NULL);
  p = p->next;
  check(i < max);
  total += sizes[i];
 }
```

One check will be recorded 1/50 times, the other not at all.

Sampling Blocks

Solution 2: Use a random number generator

```
{
   if(rand(100) == 0)check (p != NULL);
   p = p->next;
   if(rand(100) == 0) check (i < max);
   total += sizes[i];
}
```

Problem?

Sampling Blocks

Solution 2: Use a random number generator

```
{
  if(rand(100) == 0)check (p != NULL);
  p = p->next;
  if(rand(100) == 0) check (i < max);
  total += sizes[i];
}
```

Problem?

Call to rand is more expensive than the checks!

Solution: Bernoulli

- Randomized global countdown
- Selected from *geometric distribution*
  - Simulates many tosses of a biased coin
  - Stores: How many tails before next head?
    - i.e., how many sampling points to skip before we write down the next piece of data?
- Mean of distribution = expected sample rate

Solution: Bernoulli

Requires two versions of code:

- Slow path:
  - Code with the sampled instrumentation

- Fast path:
  - Code w/o the sampled instrumentation

Sampling Blocks

Fast Path Code

```
if (countdown > 2) {
    p = p->next;
    total += sizes[i];
}
```

Slow Path Code

```
if( countdown-- == 0 ) {
check(p != NULL);
countdown = getNextCountdown();
}
p = p->next;
if( countdown-- == 0 ) {
check( i < max );
countdown = getNextCountdown();
}
total += sizes[i];
```

Sampling Functions

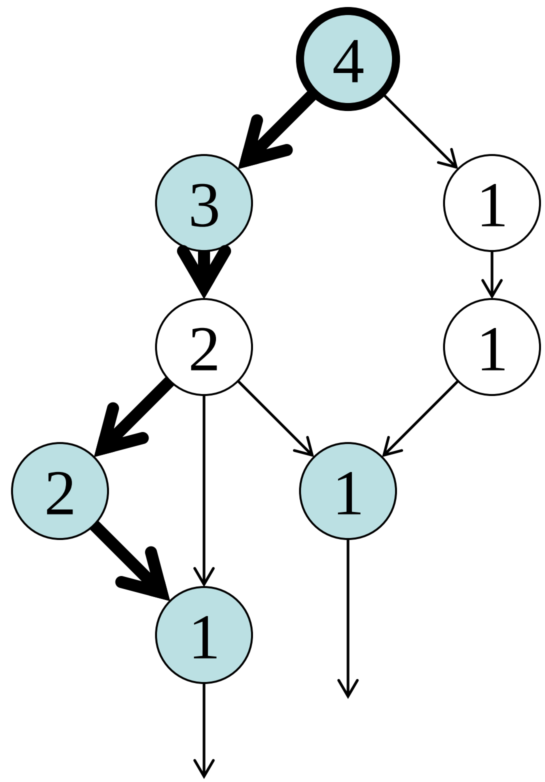
- Represent sampling blocks as a CFG
- Weight of path is the maximum number of instrumentation sites
- Place a countdown threshold check on each acyclic region
- For each region r:

  If (next-sample countdown >weight)

      no samples taken

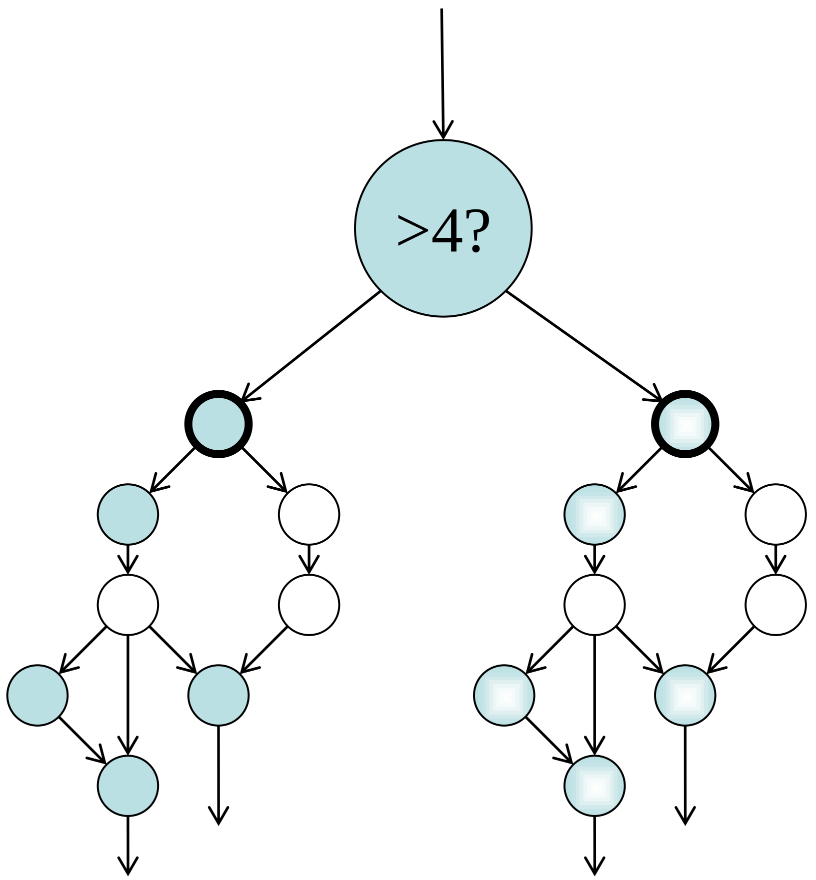
Amortized Coin Tossing

- Each acyclic region:
  - Finite number of paths
  - Finite max number of instrumentation sites
  - Shaded nodes represent instrumentation sites



Amortized Coin Tossing

- Each acyclic region:
  - Finite number of paths
  - Finite max number of instrumentation sites
- Clone each region
  - “Fast” variant
  - “Slow” sampling variant
- Choose at run time



Optimizations

- Cache global countdown in local variable
  - Global → local at func entry & after each call
  - Local → global at func exit & before each call
- Identify and ignore “weightless” functions
- Avoid cloning
  - Instrumentation-free prefix or suffix
  - Weightless or singleton regions
- Static branch prediction at region heads
- Partition sites among several binaries
- Many additional possibilities …

Colleges and Universities

This New York University is named for the family whose company was the first to sell toothpaste in a tube

Sharing the Cost of Assertions

Now we know how to sample things.

Does this work in practice?

- Let's do a series of experiments.

First: microbenchmark for sampling costs!

- What to sample: `assert()` statements
- Identify (for debugging) assertions that
  - *Sometimes fail* on bad runs
  - But *always succeed* on good runs

Case Study: CCured Safety Checks

Assertion-dense C code

Worst-case scenario for us

- Each assertion extremely fast

No bugs here; purely performance study

• Unconditional:	55% average overhead
• $^1/_{100}$ sampling:	17% average overhead
• $^1/_{1000}$ sampling:	10% average; half below 5%

Effectiveness of Sampling

- At density 1/1000 for observing rare program behavior?
  - To achieve confidence level =90%,
  - Least number of runs needed = 230,258 !
  - Solution: No. of licensed Office XP users = 16 million

2 runs/week/user = 230258 runs every 19 min!

Isolating a Deterministic Bug

- Guess predicates on scalar function returns

  `(f() < 0)`   `(f() == 0)`   `(f() > 0)`

- Count how often each predicate holds
  - Client-side reduction into counter triples
- Identify differences in good versus bad runs
  - Predicates *observed true* on some bad runs
  - Predicates *never observed true* on any good run

Function return triples aren't the only things we can sample.

Case Study: `ccrypt` Crashing Bug

- 570 call sites
- 3 × 570 = 1710 counters

Simulate large user community

- 2990 randomized runs; 88 crashes

Sampling density $^{1}/_{1000}$

- Less than 4% performance overhead

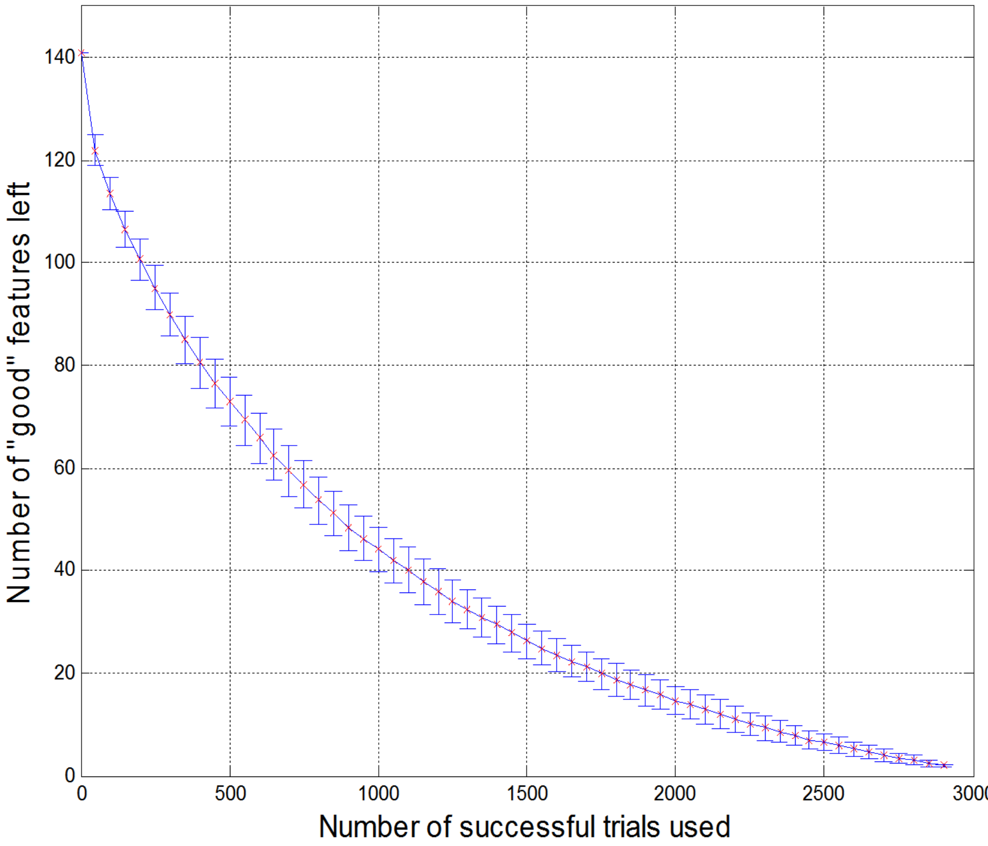
Recall goal: sampled predicates should make it easier to debug the code …

Winnowing Down to the Culprits

- 1710 counters
- 1569 are always zero
  - 141 remain
- 139 are nonzero on some successful run
- Not much left!

```
file_exists() > 0
xreadline() == 0
```

How do these pin down the bug? You'll see in a second.



Isolating a Non-Deterministic Bug

- Guess: at each direct scalar assignment

  `x = …`

- For each same-typed in-scope variable `y`
- Guess predicates on `x` and `y`

  `(x < y)`   `(x == y)`   `(x > y)`

- Count how often each predicate holds
  - Client-side reduction into counter triples

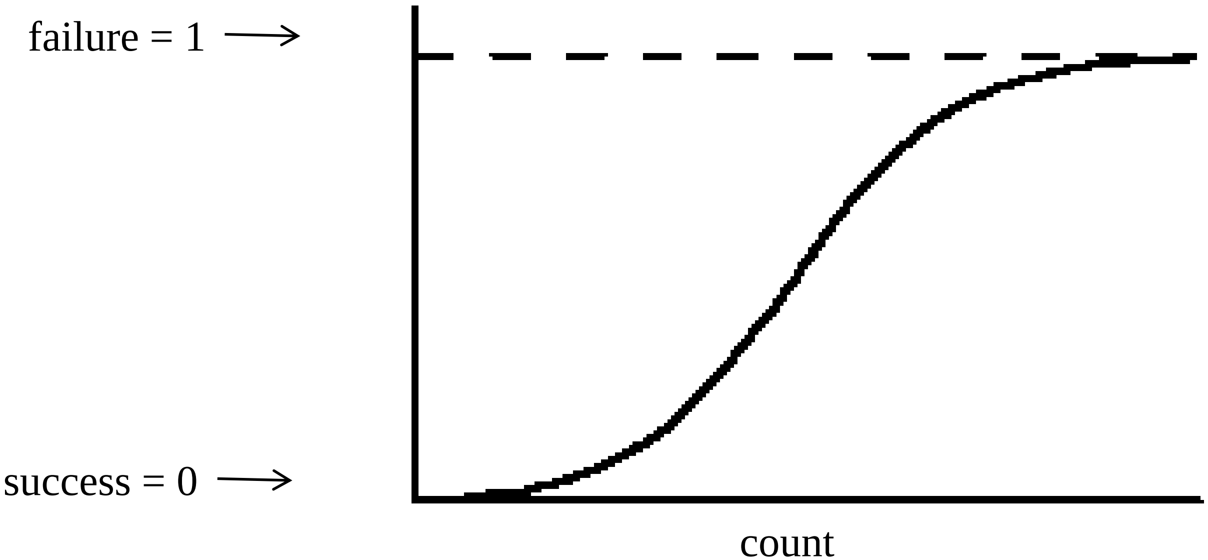
Case Study: **bc** Crashing Bug

Hunt for intermittent crash in **bc-1.06**

- Stack traces suggest heap corruption

- 2729 runs with 9MB random inputs
- 30,150 predicates on 8910 lines of code
- Sampling key to performance
  - 13% overhead without sampling
  - 0.5% overhead with $^{1}/_{1000}$ sampling

Statistical Debugging via Regularized Logistic Regression



- S-shaped cousin to linear regression
- Predict success/failure as function of counters
- Penalty factor forces most coefficients to zero
  - Large coefficient $\Rightarrow$ highly predictive of failure

Top-Ranked Predictors

```
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`

Top-Ranked Predictors

```
void more_arrays ()
{
  …

  /* Copy the old arrays. */
  for (indx = 1; indx < old_count; i
    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`

Bug Found: Buffer Overrun

```
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;

  …
}
```

It Works!

…for programs with just one bug.

- Need to deal with multiple bugs
  - How many? Nobody knows!
- Redundant predictors remain a major problem

Goal: isolate a single "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

Ranked Predicate Selection

- Consider each predicate *P* one at a time
  - Include inferred predicates (e.g. ≤, ≠, ≥)
- How likely is failure when *P* is true?
  - (technically, when *P* is *observed* to be true)
- Multiple bugs yield multiple bad predicates

Some Definitions

$$F(P) = \#\,\text{failing runs with } |P| > 0$$

$$S(P) = \#\,\text{successful runs with } |P| > 0$$

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

Are We Done? Not Exactly!

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

Bad(f = NULL) = 1.0

• Are We Done? Not Exactly!

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

Bad(f = NULL)	= 1.0
Bad(x = 0)	= 1.0

- Predicate (x = 0) is innocent bystander
  - Program is already doomed

Fun With Multi-Valued Logic

- Identify unlucky sites on the doomed path

$$Context(P) = \frac{F(P \vee \neg P)}{S(P \vee \neg P) + F(P \vee \neg P)}$$

- Background risk of failure for reaching this site, regardless of predicate truth/falsehood

Isolate the Predictive Value of *P*

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

$$Increase(P) = Bad(P) - Context(P)$$

- Formal correspondence to *likelihood ratio testing*

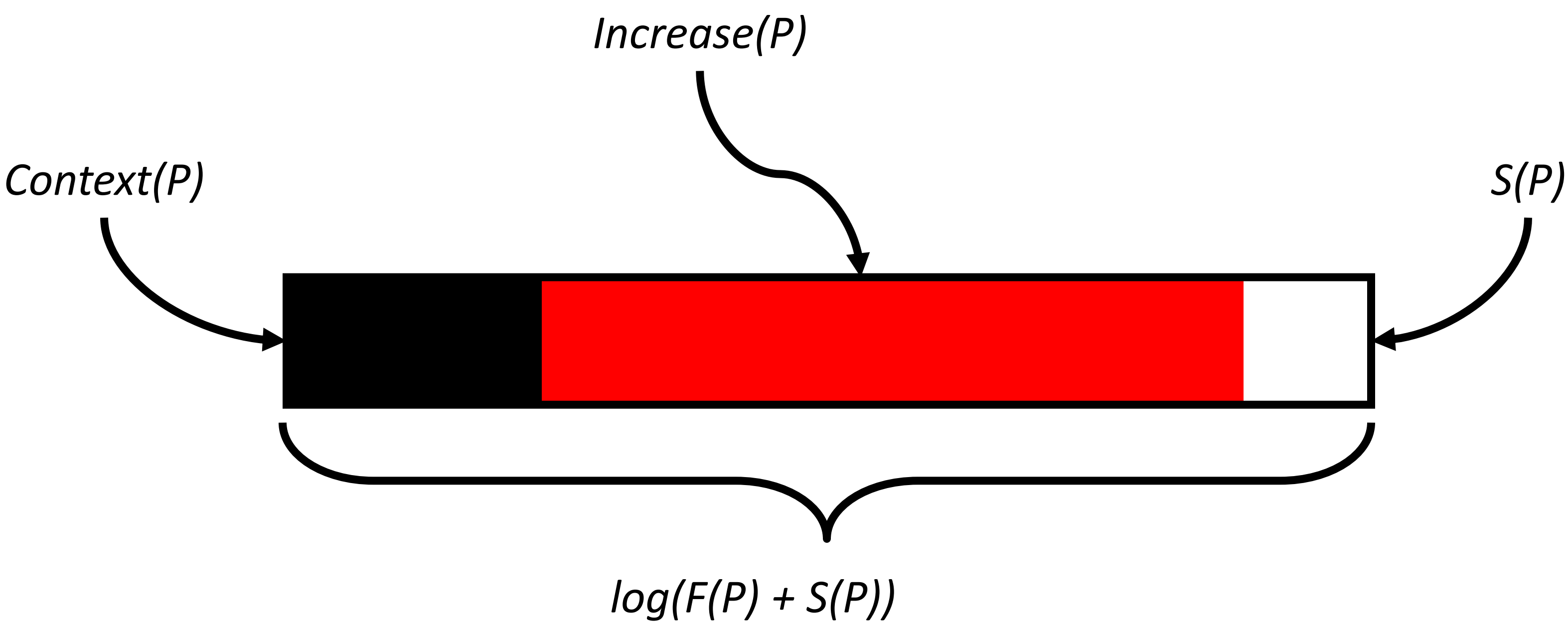
Increase Isolates the Predictor

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

Increase(f = NULL)	= 1.0
Increase(x = 0)	= 0.0

Guide to Visualization

- Multiple interesting & useful predicate metrics
- Simple visualization may help reveal trends



Bad Idea #1: Rank by *F(P)*

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.007 ± 0.012	22554	5045	27599	`files[filesindex].language != 15`
	0.176	0.007 ± 0.012	22566	5045	27611	`tmp == 0 is FALSE`
	0.176	0.007 ± 0.012	22571	5045	27616	`strcmp != 0`
	0.176	0.007 ± 0.013	18894	4251	23145	`tmp == 0 is FALSE`
	0.176	0.007 ± 0.013	18885	4240	23125	`files[filesindex].language != 14`
	0.176	0.008 ± 0.013	17757	4007	21764	`filesindex >= 25`
	0.177	0.008 ± 0.014	16453	3731	20184	`new value of M < old value of M`
	0.176	0.261 ± 0.023	4800	3716	8516	`config.winnowing_window_size != argc`
.... 2732 additional predictors follow						

- Many failing runs but low *Increase*
- Tend to be *super-bug predictors*
  - Each covers several bugs, plus lots of junk

Bad Idea #2: Rank by *Increase(P)*

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.065	0.935 ± 0.019	0	23	23	`((*(fi + i)))->this.last_token < filesbase`
	0.065	0.935 ± 0.020	0	10	10	`((*(fi + i)))->other.last_line == last`
	0.071	0.929 ± 0.020	0	18	18	`((*(fi + i)))->other.last_line == filesbase`
	0.073	0.927 ± 0.020	0	10	10	`((*(fi + i)))->other.last_line == yy_n_chars`
	0.071	0.929 ± 0.028	0	19	19	`bytes <= filesbase`
	0.075	0.925 ± 0.022	0	14	14	`((*(fi + i)))->other.first_line == 2`
	0.076	0.924 ± 0.022	0	12	12	`((*(fi + i)))->this.first_line < nid`
	0.077	0.923 ± 0.023	0	10	10	`((*(fi + i)))->other.last_line == yy_init`

.... 2732 additional predictors follow ....

- High *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

A Helpful Analogy

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

Rank by Harmonic Mean

Thermometer	Context	Increase	S	F	F + S	Predicate
	0.176	0.824 ± 0.009	0	1585	1585	`files[filesindex].language > 16`
	0.176	0.824 ± 0.009	0	1584	1584	`strcmp > 0`
	0.176	0.824 ± 0.009	0	1580	1580	`strcmp == 0`
	0.176	0.824 ± 0.009	0	1577	1577	`files[filesindex].language == 17`
	0.176	0.824 ± 0.009	0	1576	1576	`tmp == 0 is TRUE`
	0.176	0.824 ± 0.009	0	1573	1573	`strcmp > 0`
	0.116	0.883 ± 0.012	1	774	775	`((*(fi + i)))->this.last_line == 1`
	0.116	0.883 ± 0.012	1	776	777	`((*(fi + i)))->other.last_line == yyleng`
.... 2732 additional predictors follow						

- Definite improvement
  - Large increase, many failures, few or no successes
- But redundancy is *still* a problem

Redundancy Elimination

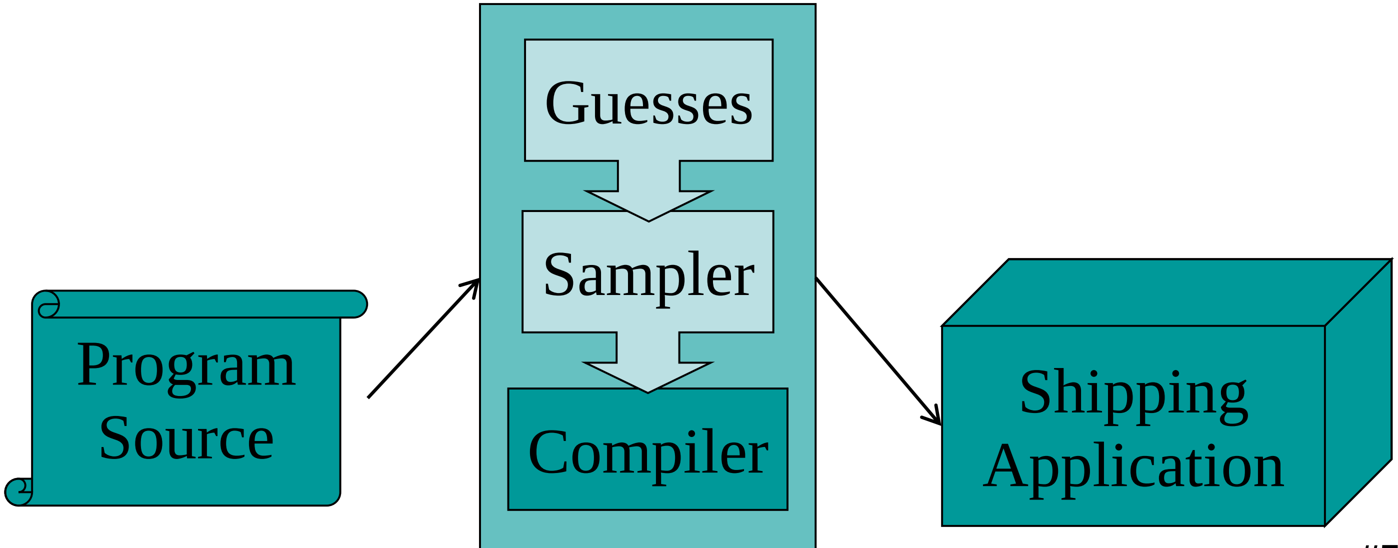
- One predictor for a bug is interesting
  - Additional predictors are a distraction
  - Want to explain each failure once
- Similar to minimum set-cover problem
  - Cover all failed runs with subset of predicates
  - Greedy selection using harmonic ranking

Moving To The Real World

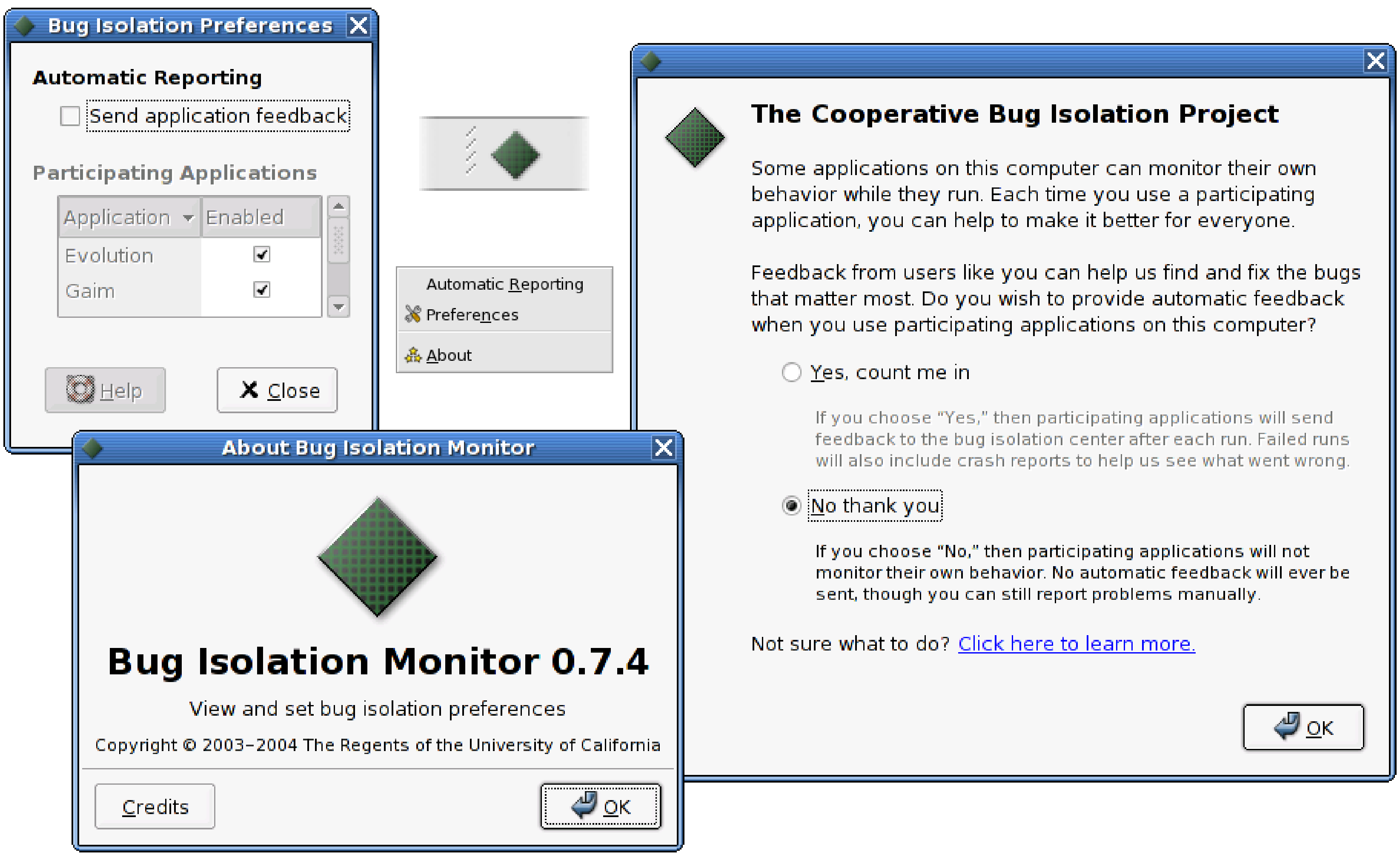
- Pick instrumentation scheme
- Automatic tool instruments program
- Sampling yields low overhead
- Many users run program
- Many reports) find bug
- So let's do it!

Native Compiler Integration

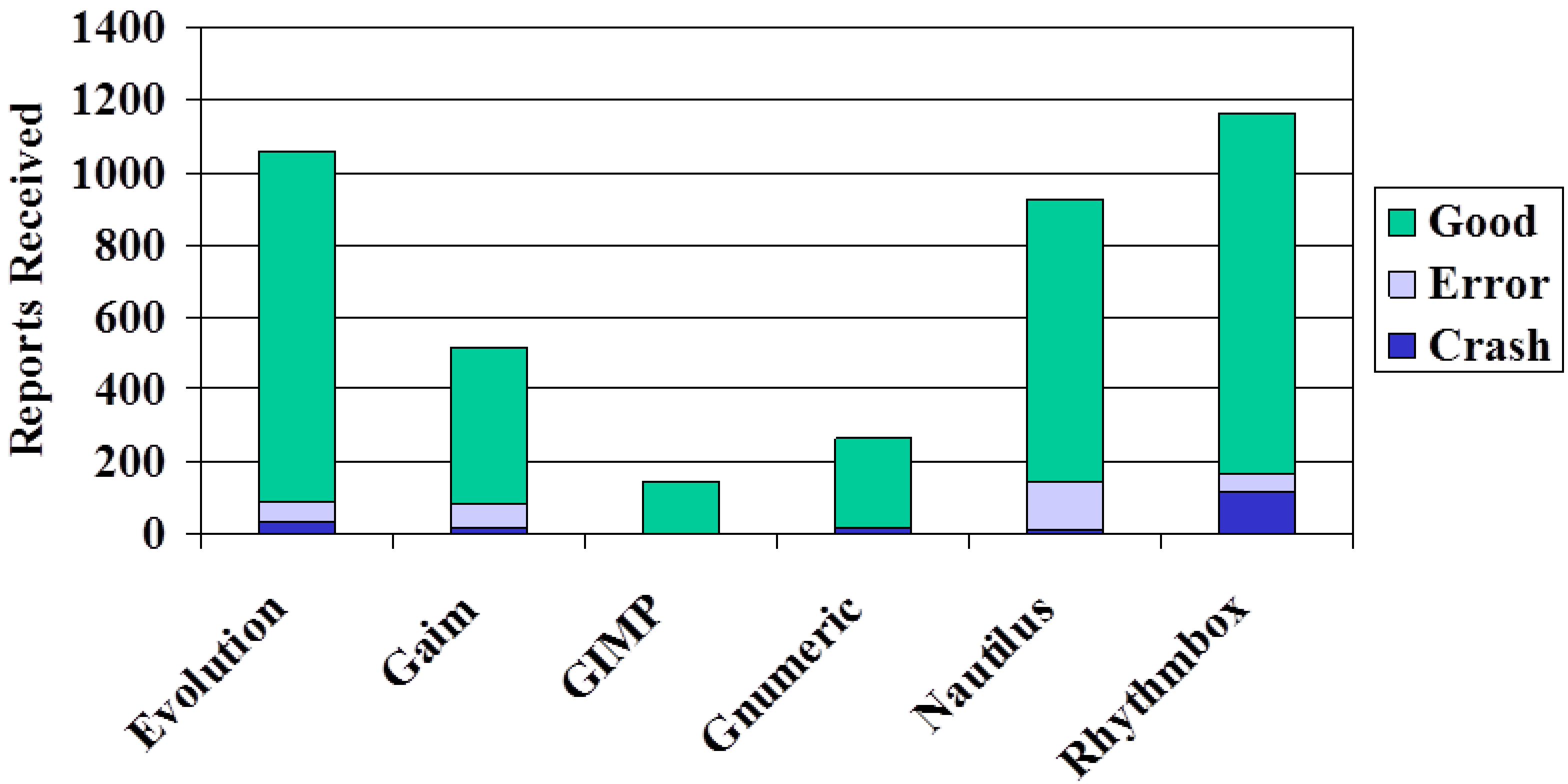
- Instrumentor must mimic native compiler
  - You don't have time to port & annotate by hand
- This approach: source-to-source, then native
- Hooks for GCC:



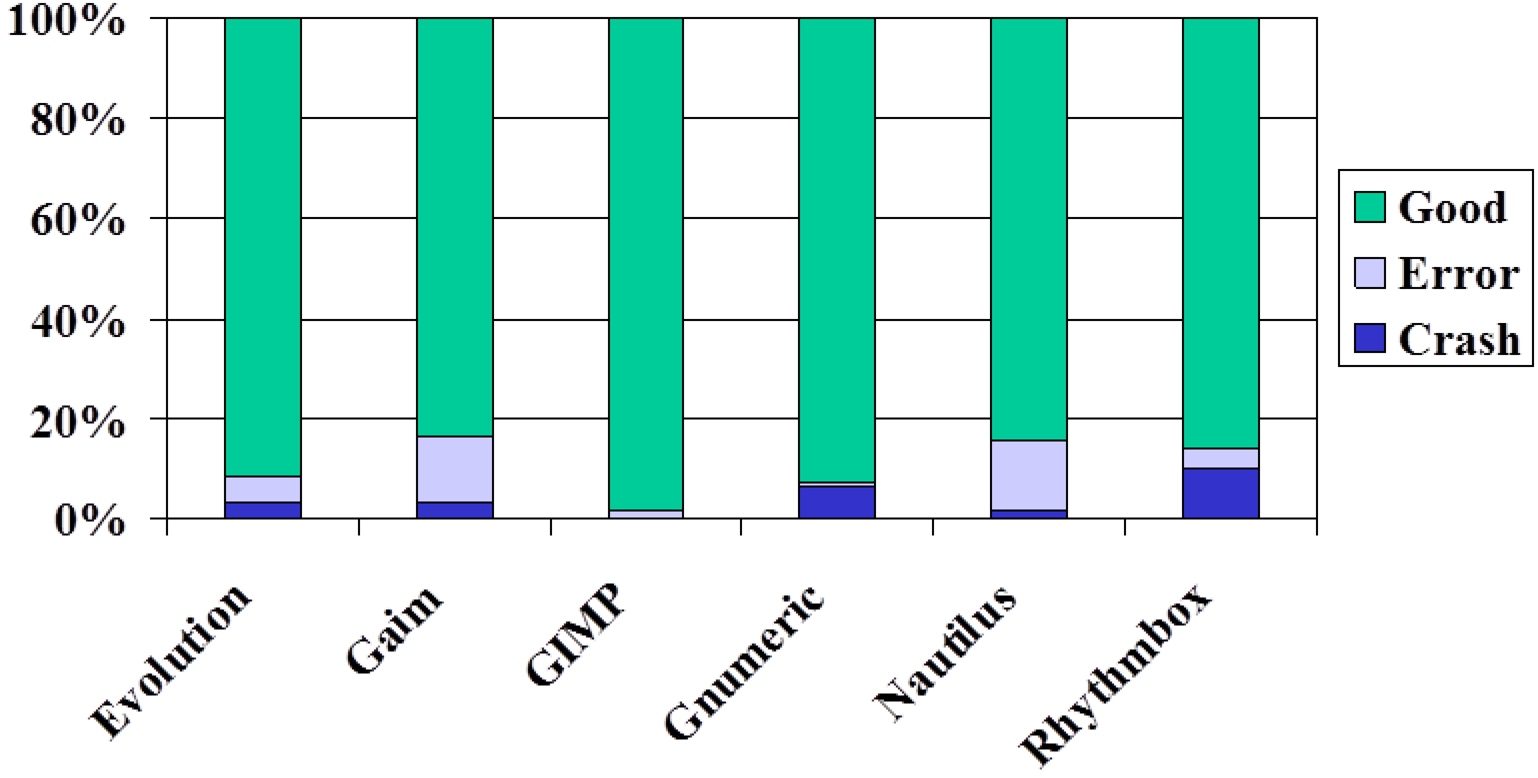
Keeping the User In Control



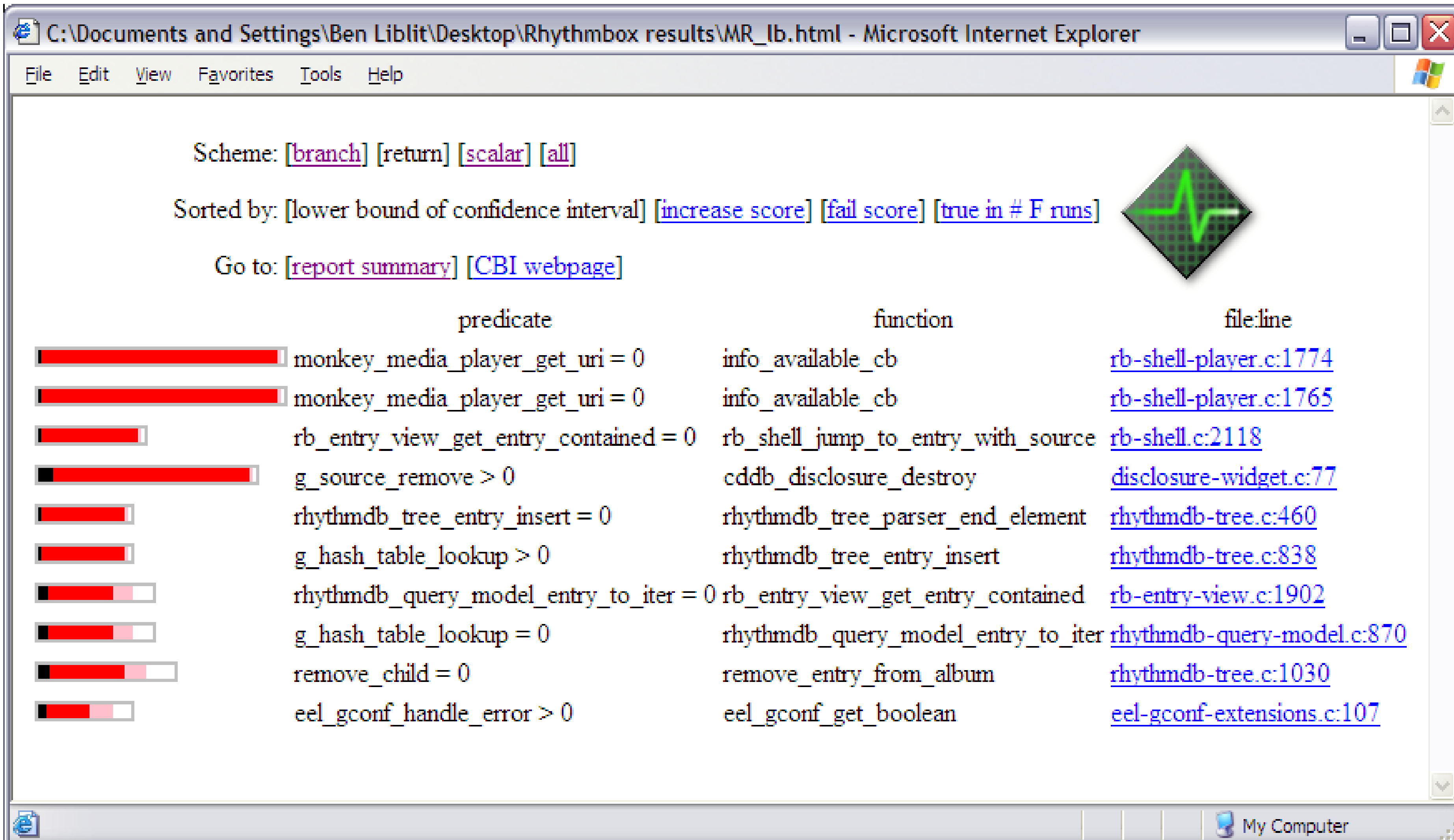
Public Deployment 2004



Public Deployment 2004



Sneak Peak: Data Exploration


C:\Documents and Settings\Ben Liblit\Desktop\Rhythmbox results\MR_lb.html - Microsoft Internet Explorer

File Edit View Favorites Tools Help

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

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

Go to: [report summary] [CBI webpage]

predicate	function	file:line
monkey_media_player_get_uri = 0	info_available_cb	rb-shell-player.c:1774
monkey_media_player_get_uri = 0	info_available_cb	rb-shell-player.c:1765
rb_entry_view_get_entry_contained = 0	rb_shell_jump_to_entry_with_source	rb-shell.c:2118
g_source_remove > 0	cddb_disclosure_destroy	disclosure-widget.c:77
rhythmdb_tree_entry_insert = 0	rhythmdb_tree_parser_end_element	rhythmdb-tree.c:460
g_hash_table_lookup > 0	rhythmdb_tree_entry_insert	rhythmdb-tree.c:838
rhythmdb_query_model_entry_to_iter = 0	rb_entry_view_get_entry_contained	rb-entry-view.c:1902
g_hash_table_lookup = 0	rhythmdb_query_model_entry_to_iter	rhythmdb-query-model.c:870
remove_child = 0	remove_entry_from_album	rhythmdb-tree.c:1030
eel_gconf_handle_error > 0	eel_gconf_get_boolean	eel-gconf-extensions.c:107

My Computer

Summary: Putting it All Together

- Flexible, fair, low overhead sampling
- Predicates probe program behavior
  - Client-side reduction to counters
  - Most guesses are uninteresting or meaningless
- Seek behaviors that co-vary with outcome
  - Deterministic failures: process of elimination
  - Non-deterministic failures: statistical modeling

Conclusions

- Bug triage that directly reflects *reality*
  - Learn the most, most quickly, about the bugs that happen most often
- Variability is a benefit rather than a problem
  - Results grow stronger over time
- Find bugs while you sleep!
- Public deployment is challenging
  - Real world code pushes tools to their limits
  - Large user communities take time to build

Homework

- Projects!