# Cooperative Bug Isolation

Ben Liblit *et al*.


## What's This?

- I decided that that sigma calculus for objects was "too heavy" for our final lecture.
- OO slides are available on the webpage.
- Instead, we'll talk about the work that won the 2005 ACM Doctoral Dissertation Award.

## Sic Transit Gloria Raymondi

- 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.

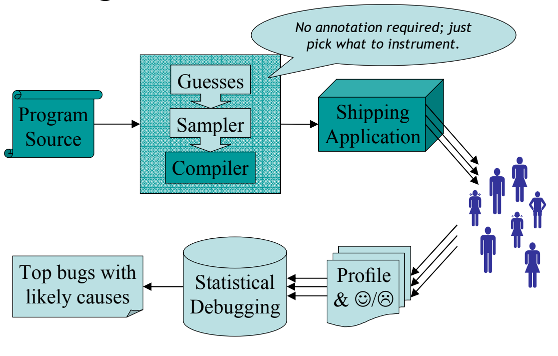


## Today's Goal: Measure Reality

- We measure bridges, airplanes, cars…
  - Where is flight data recorder for software?
- Users are a vast, untapped resource
  - 60 million 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



## Bug Isolation Architecture






## Why Will This Work?

- Good News: Users can help!
- Important bugs happen often, to many users
  - User communities are big and growing fast
  - User runs á testing runs
  - Users are networked
- We *can* do better, with help from users!
  - cf. crash reporting (Microsoft, Netscape)
  - Today: research efforts

## Let's Use Randomness


- Problem: recording everything is too expensive!
- Idea: each user records 0.1% of everything
- Generic sparse sampling framework
  - Adaptation of Arnold & Ryder
- Suite of instrumentations / analyses
  - Sharing the cost of assertions
  - Isolating deterministic bugs
  - Isolating non-deterministic bugs



## Sampling the Bernoulli Way

- Identify the points of interest
- Decide to examine or ignore each site…
  - Randomly
  - Independently
  - Dynamically
- × Cannot use clock interrupt: no context
- × Cannot be periodic: unfair
- × Cannot toss coin at each site: too slow



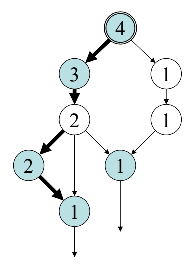
## Anticipating the Next Sample

- Randomized global countdown
- Selected from *geometric distribution*
  - Inter-arrival time for biased coin toss
  - 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



## Amortized Coin Tossing

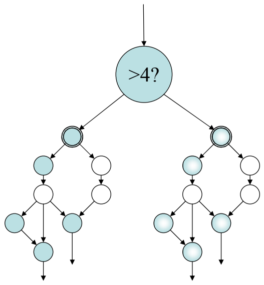
- Each acyclic region:
  - Finite number of paths
  - Finite max number of instrumentation sites
  - Shaded nodes represent instrumentation sites X






## 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 …

## 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%



## 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 …

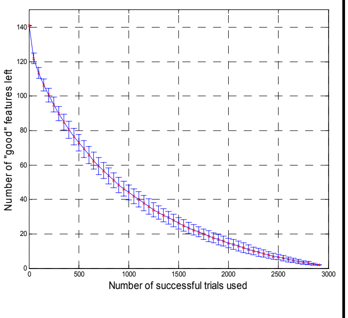


## Winnowing Down to the Culprits

- 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

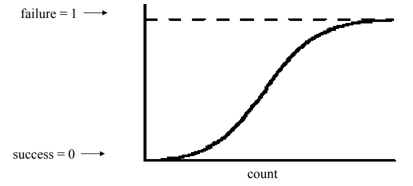
  **`x = …`**
- For each same-typed in-scope variable **`y`**
- Guess predicates on **`x`** and **`y`**

  **`(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 ⇒ highly predictive of failure



# Top-Ranked Predictors

```
void more_arrays ()
{
  …

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

  …
}
```



# 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!


# Multithreaded Programs

- Global next-sample countdown
  - High contention, small footprint
  - Want to use registers for performance
  - ⇒Thread-local: one countdown per thread
- Global predicate counters
  - Low contention, large footprint
  - ⇒Optimistic atomic increment

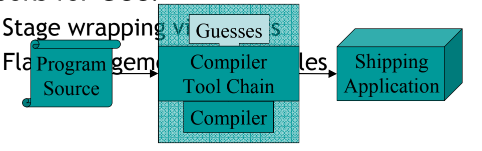
# Multi-Module Programs

- Forget about global static analysis
  - Plug-ins, shared libraries
  - Instrumented & uninstrumented code
- Self-management at compile time
  - Locally derive identifying object signature
  - Embed static site information within object file
- Self-management at run time
  - Report feedback state on normal object unload
  - Signal handlers walk global object registry



# 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:
  - Stage wrapping vi… s
  - Fla… gem… les

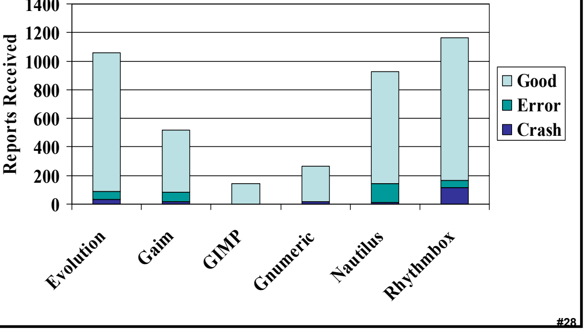





# Keeping the User In Control

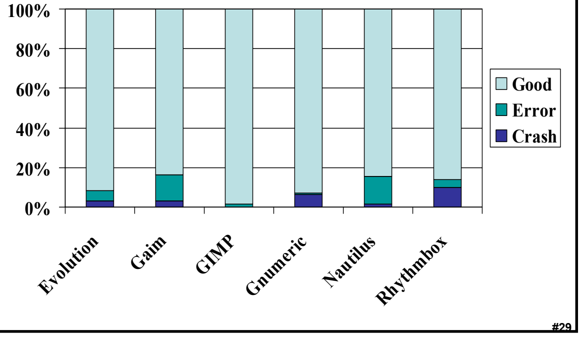



# Public Deployment 2004



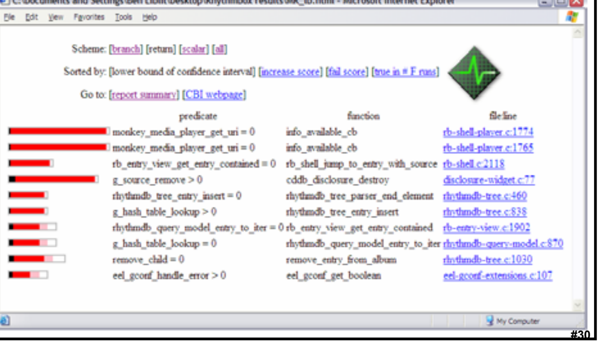



# Public Deployment 2004






# Sneak Peak: Data Exploration



## 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
- But the results are worth it:

*"Thanks to Ben Liblit and the Cooperative Bug Isolation Project, this version of Rhythmbox should be the most stable yet."*



## Homework

- Good luck with your project presentations!
- Have a lovely summer.