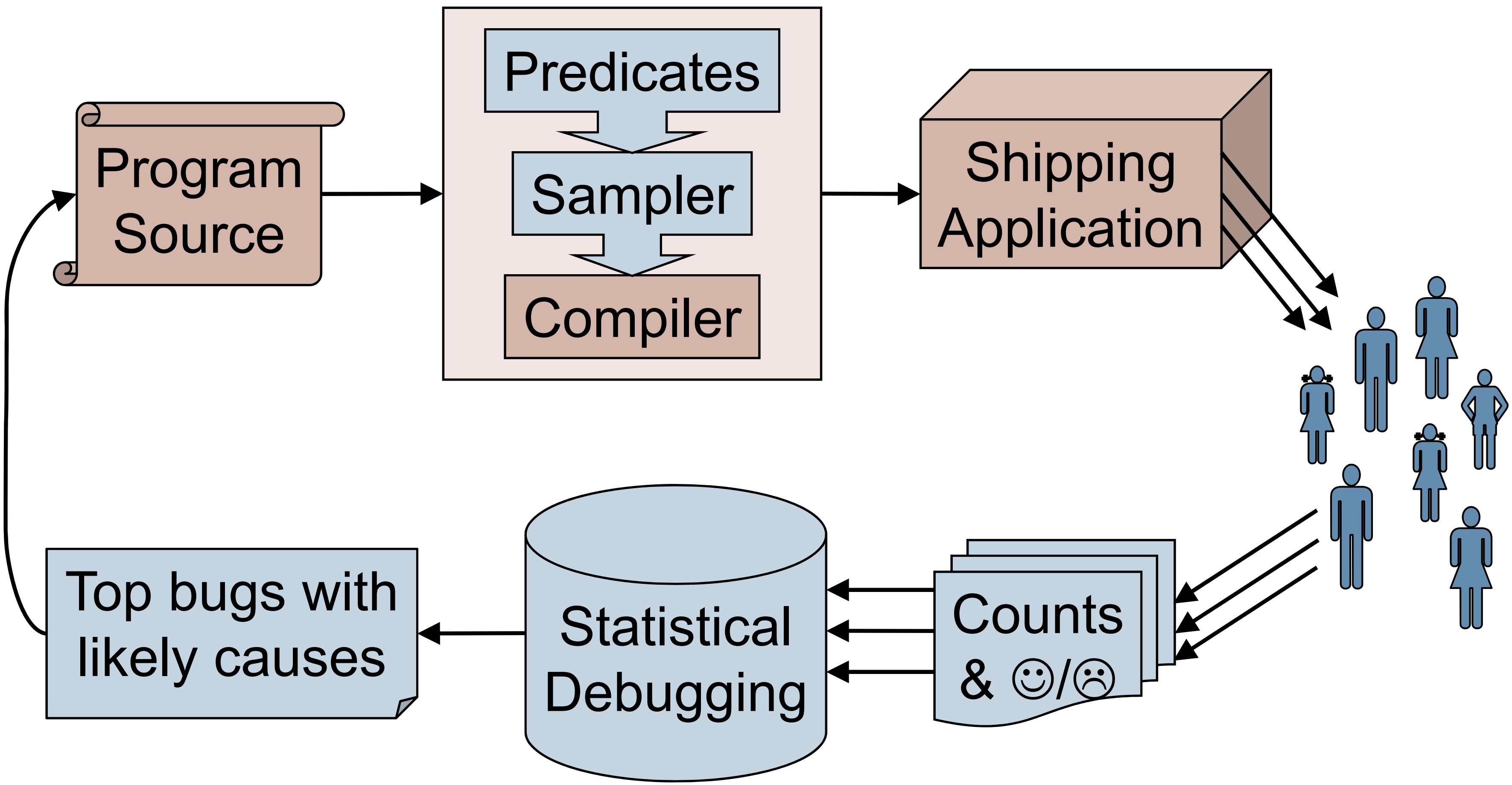
Statistical Debugging

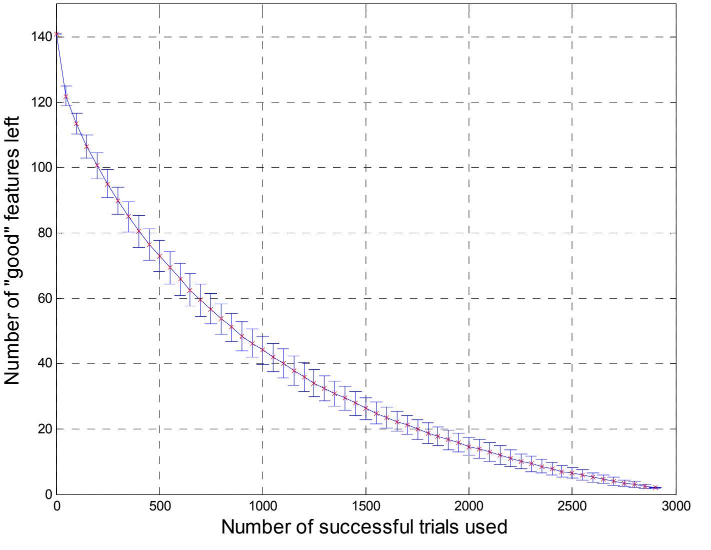
Ben Liblit, University of Wisconsin–Madison

Bug Isolation Architecture



Winnowing Down the Culprits

- 1710 counters
  - 3 × 570 call sites
- 1569 zero on all runs
  - 141 remain
- 139 nonzero on at least one successful run
- Not much left!
  - `file_exists() > 0`
  - `xreadline() == 0`



Multiple, Non-Deterministic Bugs

- Strict process of elimination won't work
  - Can't assume program will crash when it should
  - No single common characteristic of all failures
- Look for general correlation, not perfect prediction
- ***Warning! Statistics ahead!***

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

It Works!

…for programs with just one bug.

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

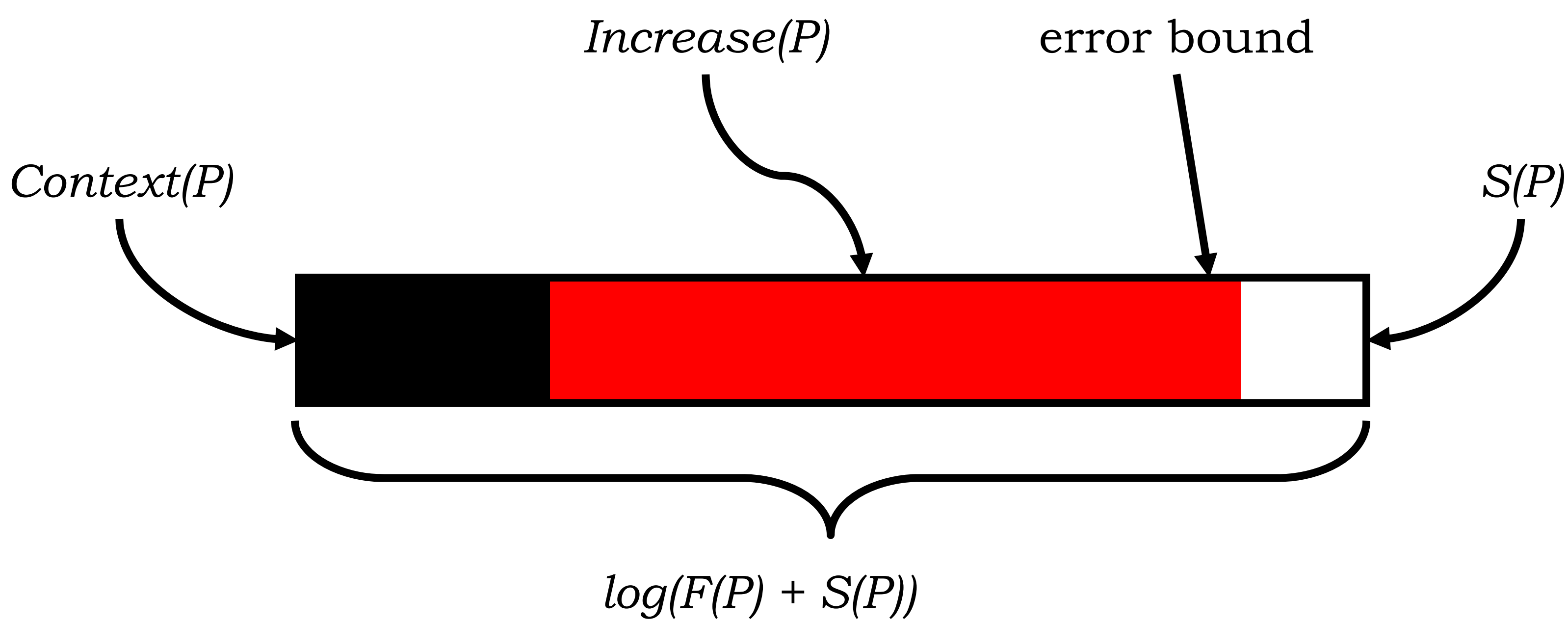
Goal: isolate a single "best" predictor for each bug, with no prior knowledge of the number of bugs.

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

Guide to Visualization

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



Confidence Interval on *Increase(P)*

$$\pm 1.96\sqrt{\frac{Bad(P)\cdot\left(1-Bad(P)\right)}{S(P)+F(P)}+\frac{Context(P)\cdot\left(1-Context(P)\right)}{S(P\vee\neg P)+F(P\vee\neg P)}}$$

- Strictly speaking, this is slightly bogus
  - *Bad(P)* and *Context(P)* are not independent
  - Correct confidence interval would be larger

Bad Idea #1: 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

Bad Idea #2: 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

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
  - 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

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

Case Study: Moss

- Reintroduce nine historic Moss bugs
  - High- and low-level errors
  - Includes wrong-output bugs
- Instrument with everything we've got
  - Branches, returns, variable value pairs, the works
- 32,000 randomized runs at $^{1}/_{100}$ sampling

Effectiveness of Filtering

Scheme	Total	Retained	Rate
branches	4170	18	0.4%
returns	2964	11	0.4%
value-pairs	195,864	2682	1.4%

Effectiveness of Ranking

- Five bugs: captured by branches, returns
  - Short lists, easy to scan
  - Can stop early if Bad drops down
- Two bugs: captured by value-pairs
  - Much redundancy
- Two bugs: never cause a failure
  - No failure, no problem
- One surprise bug, revealed by returns!

Analysis of exif

Initial	Effective	Predicate
		`i < 0`
		`maxlen > 1900`
		`o + s > buf_size is TRUE`

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

Analysis of Rhythmbox

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

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

How Many Runs Are Needed?

	Failing Runs For Bug #*n*						
	#1	#2	#3	#4	#5	#6	#9
Moss	18	10	32	12	21	11	20
ccrypt	26						
bc	40						
Rhythmbox	22	35					
exif	28	12	13				

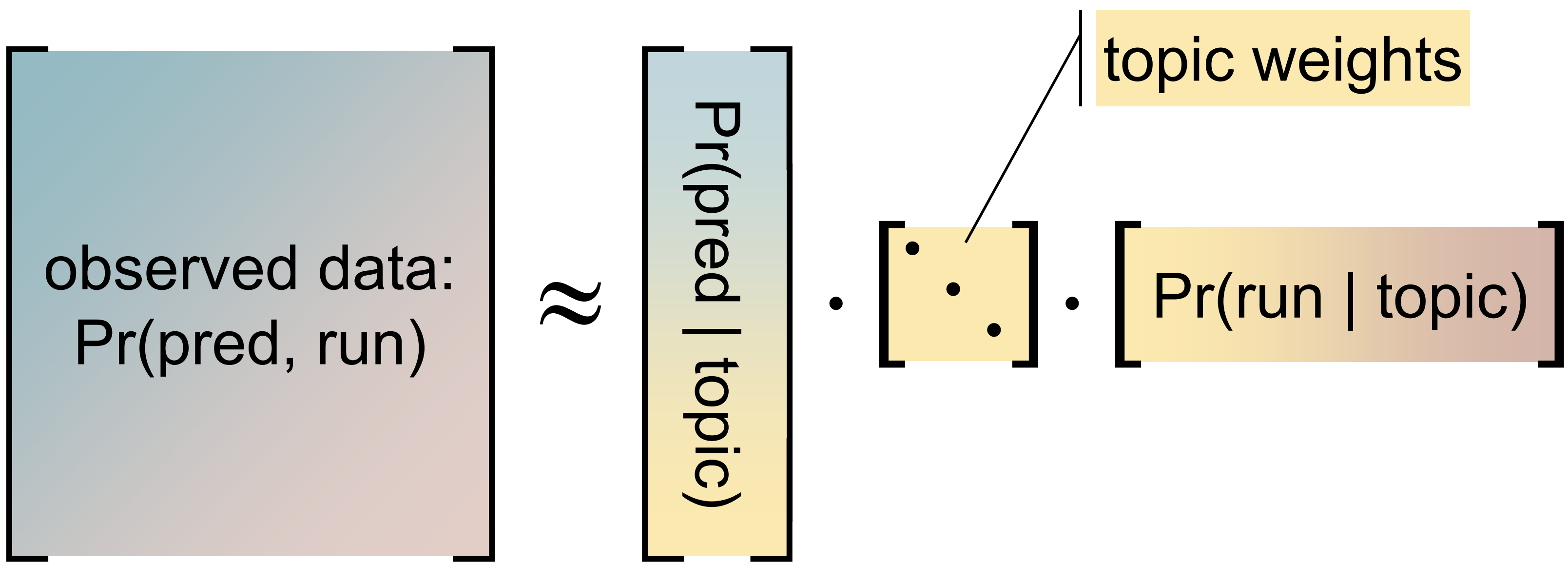
Other Models, Briefly Considered

- Regularized logistic regression
  - S-shaped curve fitting
- Bipartite graphs trained with iterative voting
  - Predicates vote for runs
  - Runs assign credibility to predicates
- Predicates as random distribution pairs
  - Find predicates whose distribution parameters differ
- Random forests, decision trees, support vector machines, …

Capturing Bugs and Usage Patterns

- Borrow from natural language processing
  - Identify topics, given term-document matrix
  - Identify bugs, given feature-run matrix
- Latent semantic analysis and related models
  - Topics ⇔ bugs and usage patterns
  - Noise words ⇔ common utility code
  - Salient keywords ⇔ buggy code

Probabilistic Latent Semantic Analysis



Uses of Topic Models

- Cluster runs by most probable topic
  - Failure diagnosis for multi-bug programs
- Characterize representative run for cluster
  - Failure-inducing execution profile
  - Likely execution path to guide developers
- Relate usage patterns to failure modes
  - Predict system (in)stability in scenarios of interest

Compound Predicates for Complex Bugs

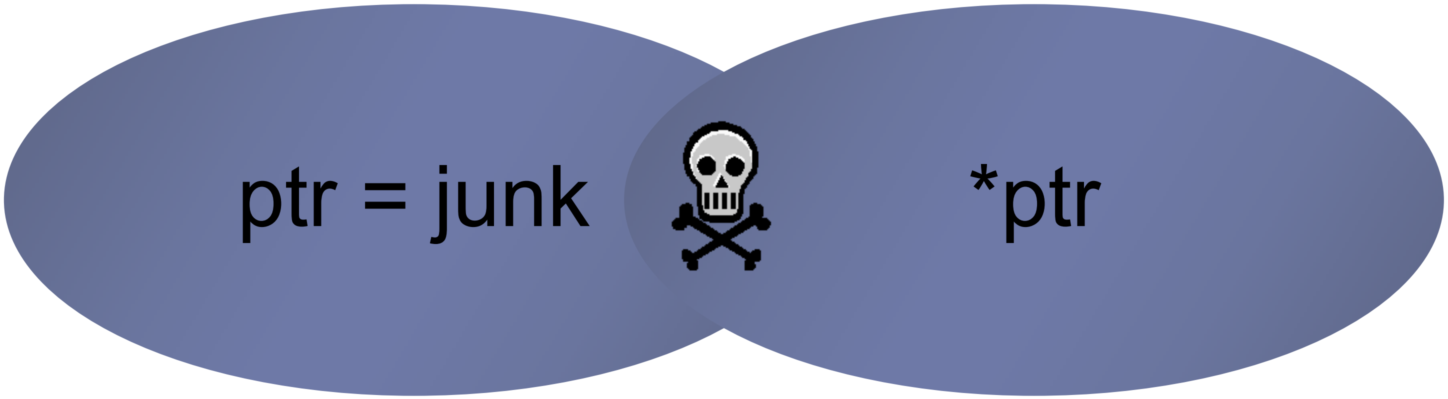
“Logic, like whiskey, loses its beneficial effect when taken in too large quantities.”

Edward John Moreton Drax Plunkett, Lord Dunsany,
“Weeds & Moss”, *My Ireland*

Limitations of Simple Predicates

- Each predicate partitions runs into 2 sets:
  - Runs where it was true
  - Runs where it was false
- Can accurately predict bugs that match this partition
- Unfortunately, some bugs are more complex
  - Complex border between good & bad
  - Requires richer language of predicates

Motivation: Bad Pointer Errors



- In function `exif_mnote_data_canon_load`:
  - ```
    for (i = 0; i < c; i++) {
        …
        n->count = i + 1;
        …
        if (o + s > buf_size) return;
        …
        n->entries[i].data = malloc(s);
        …
    }
    ```
- Crash on later use of `n->entries[i].data`
    ```

Motivation: Bad Pointer Errors

Kinds of Predicate	Best Predicate	Score
Simple Only	new `len` == old `len`	0.71
Simple & Compound	`o + s > buf_size` ∧ `offset < len`	0.94

- In function `exif_mnote_data_canon_load`:
  - ```
    for (i = 0; i < c; i++) {
        …
        n->count = i + 1;
        …
        if (o + s > buf_size) return;
        …
        n->entries[i].data = malloc(s);
        …
    }
    ```
- Crash on later use of `n->entries[i].data`
    ```

Great! So What's the Problem?

- Too many compound predicates
  - 2^{2^N} functions of N simple predicates
  - N^2 conjunctions & disjunctions of two variables
  - $N \sim 100$ even for small applications
- Incomplete information due to sampling
- Predicates at different locations

Conservative Definition

- A conjunction $C = p_1 \wedge p_2$ is true in a run iff:
  - p_1 is true at least once and
  - p_2 is true at least once
- Disjunction is defined similarly
- Disadvantage:
  - C may be true even if p_1, p_2 never true simultaneously
- Advantages:
  - Monitoring phase does not change
  - $p_1 \wedge p_2$ is just another predicate, inferred offline

Three-Valued Truth Tables

- For each predicate & run, three possibilities:
  1. True (at least once)
  2. Not true (and false at least once)
  3. Never observed

Conjunction: $p_1 \land p_2$

p_2 \ p_1	T	F	?
T	T	F	?
F	F	F	**F**
?	?	**F**	?

Disjunction: $p_1 \lor p_2$

p_2 \ p_1	T	F	?
T	T	T	**T**
F	T	F	?
?	**T**	?	?

Mixed Compound & Simple Predicates

- Compute score of each conjunction & disjunction
  - $C = p_1 \wedge p_2$
  - $D = p_1 \vee p_2$
- Compare to scores of constituent simple predicates
  - Keep if higher score: better partition between good & bad
  - Discard if lower score: needless complexity
- Integrates easily into iterative ranking & elimination

Still Too Many

- Complexity: $N^2 \cdot R$
  - N = number of simple predicates
  - R = number of runs being analyzed
  - 20 minutes for $N \sim 500$, $R \sim 5{,}000$
- Idea: early pruning optimization
- Compute upper bound of score and discard if too low
  - "Too low" = lower than constituent simple predicates
- Reduce $O(R)$ to $O(1)$ per complex predicate

Upper Bound On Score

- $\uparrow$ Harmonic mean

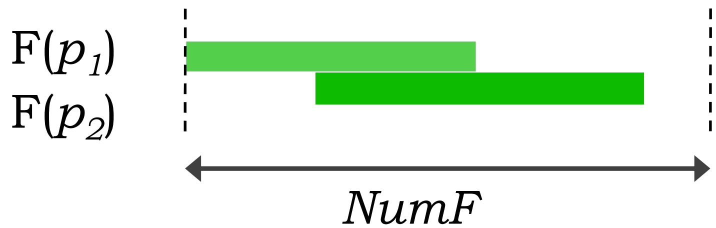
$$\uparrow Increase(C) = \frac{\uparrow F(C)}{\uparrow F(C) + \downarrow S(C)} - \frac{\downarrow F(C\ \text{obs})}{\downarrow F(C\ \text{obs}) + \uparrow S(C\ \text{obs})}$$

$$\uparrow Sensitivity(C) = \frac{\uparrow \log F(C)}{\log NumF}$$

- Upper Bound on $C = p_1 \wedge p_2$
  - Find $\uparrow F(C)$, $\downarrow S(C)$, $\downarrow F(C\ \text{obs})$ and $\uparrow S(C\ \text{obs})$
  - In terms of corresponding counts for p_1, p_2

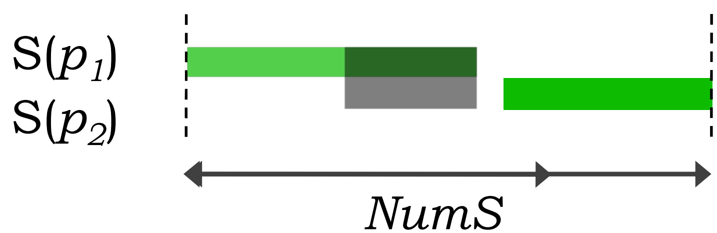
↑F(C) and ↓S(C) for conjunction

- ↑F(C): true runs completely overlap



Min(F(p_1), F(p_2))

- ↓S(C): true runs are disjoint

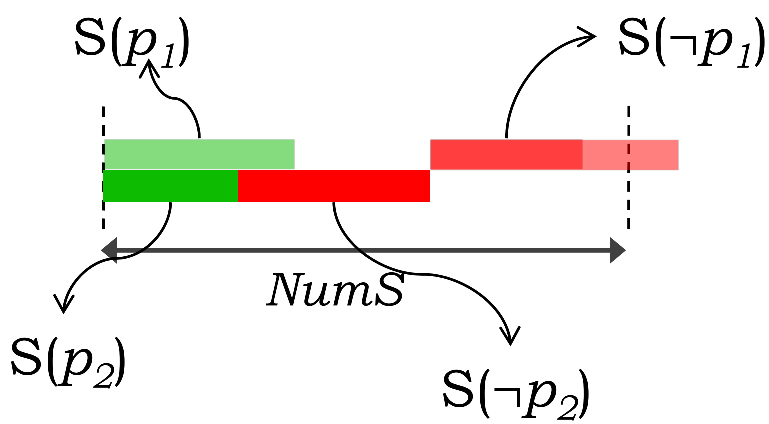


either 0 or
S(p_1)+S(p_2) - *NumS*

(whichever is maximum)

↑S(C obs)

- Maximize two cases
- C = true
  - True runs of p_1, p_2 overlap
- C = false
  - False runs of p_1, p_2 are disjoint



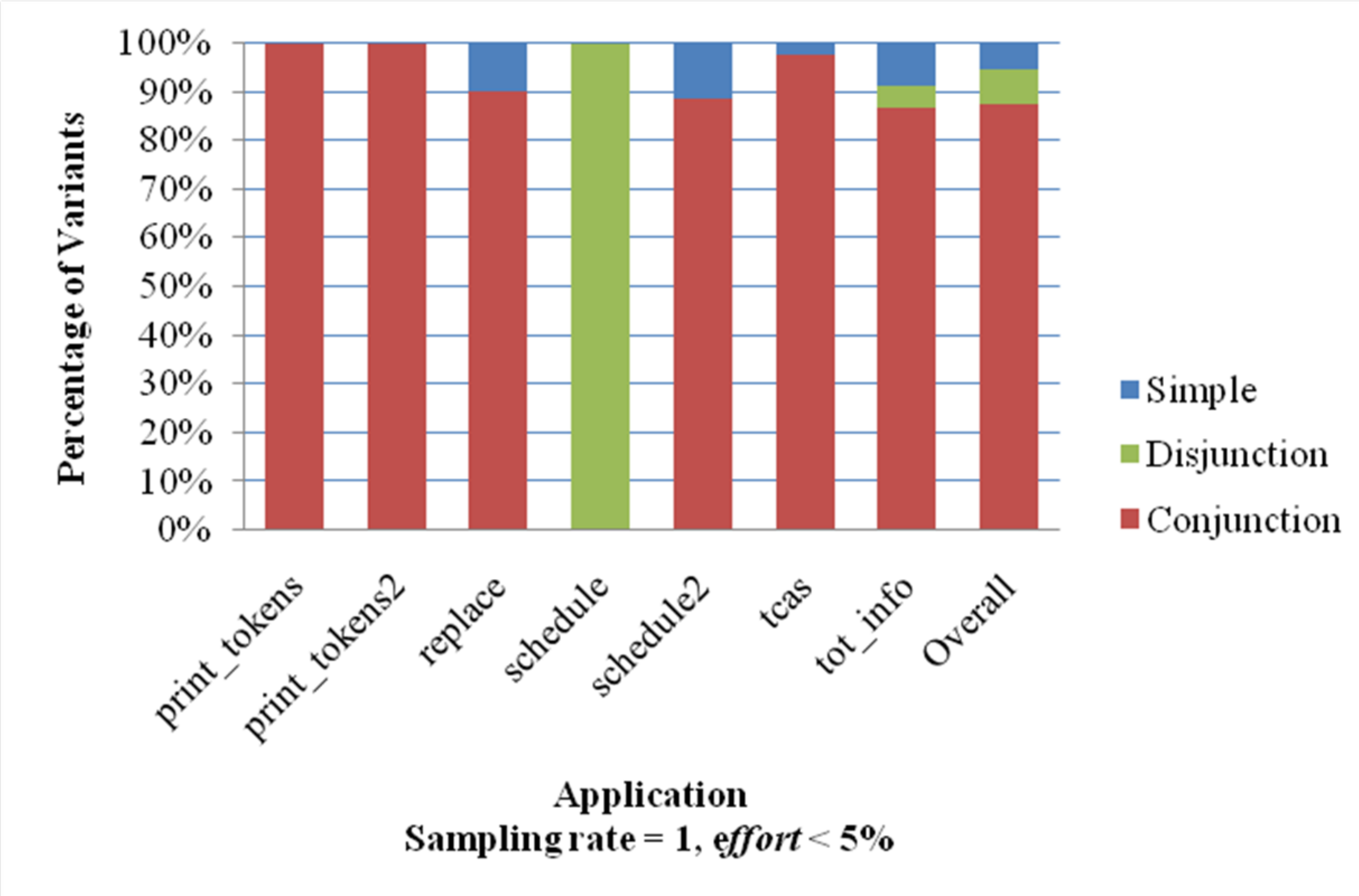
Min(S(p_1), S(p_2)) + S($\neg p_1$) + S($\neg p_2$)
or
NumS

(whichever is minimum)

Usability

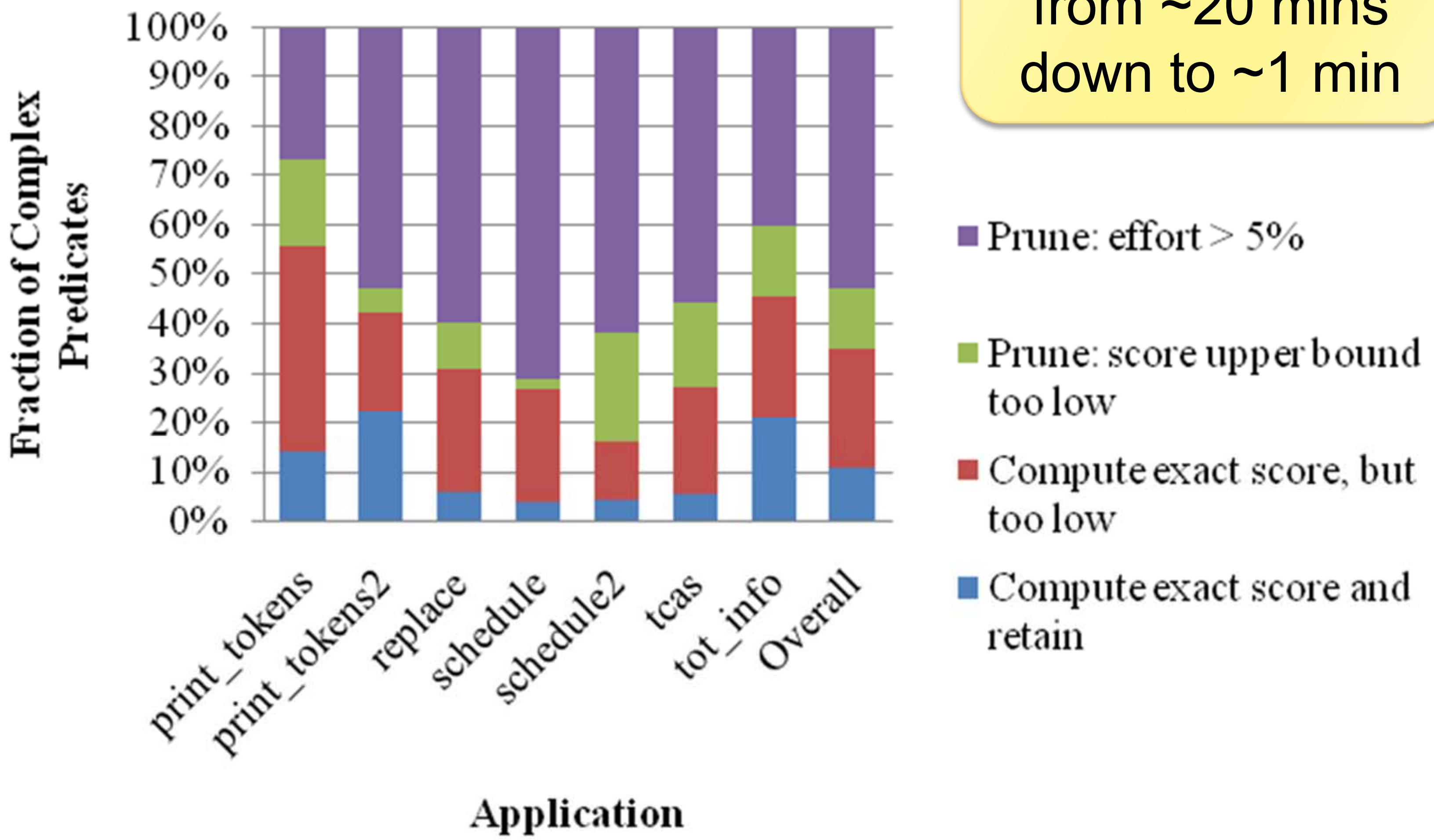
- Complex predicates can confuse programmer
  - Non-obvious relationship between constituents
  - Prefer to work with easy-to-relate predicates
- $effort(p_1, p_2)$ = proximity of p_1 and p_2 in PDG
  - PDG = CDG ∪ DDG
  - Per Cleve and Zeller [ICSE '05]
- Fraction of entire program
  - "Usable" only if *effort* < 5%
  - Somewhat arbitrary cutoff; works well in practice

Evaluation:
What kind of predicate has the top score?

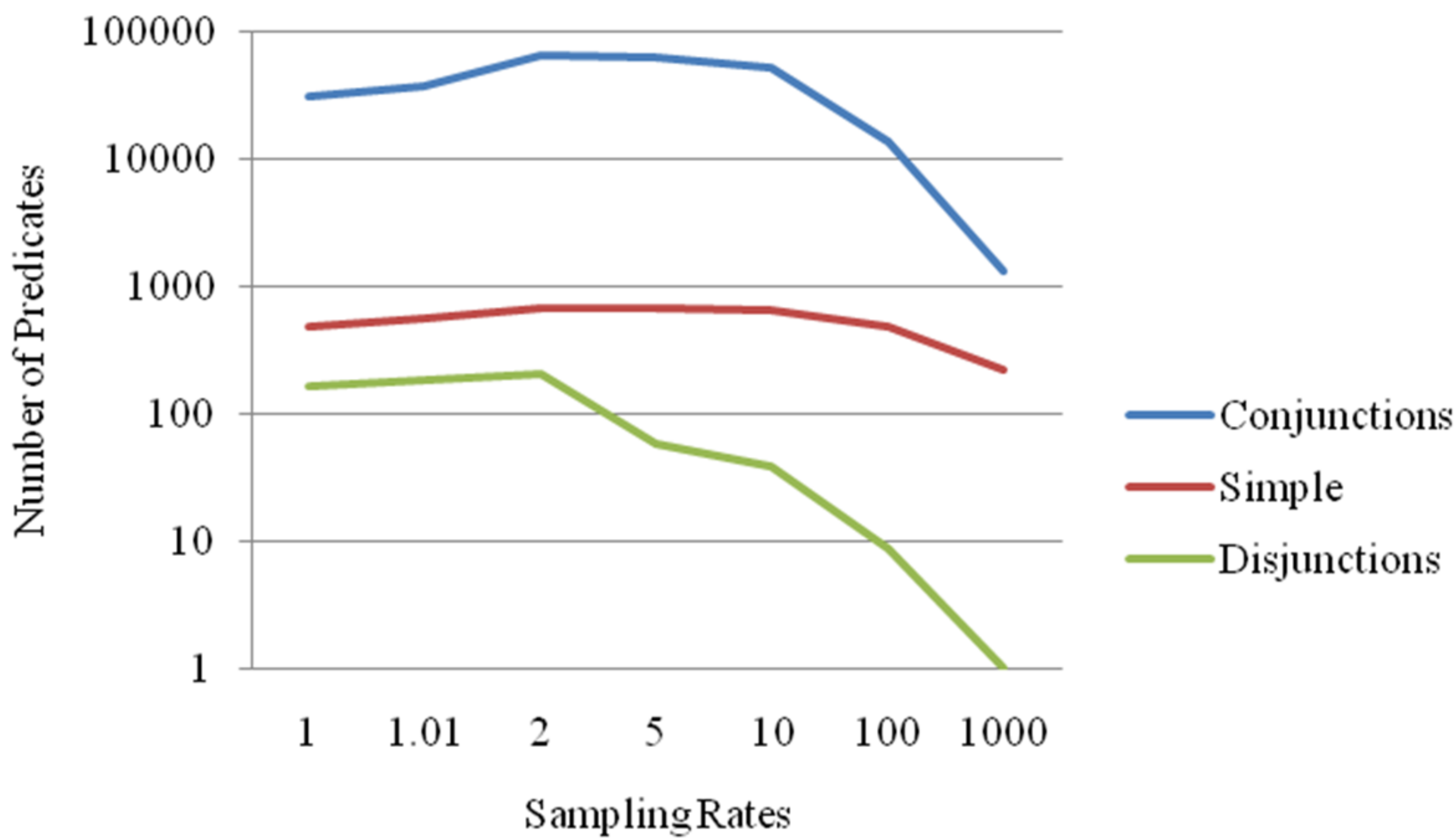


Evaluation: Effectiveness of Pruning

Analysis time: from ~20 mins down to ~1 min



Evaluation: Impact of Sampling



Application: print_tokens

Evaluation: Usefulness Under Sparse Sampling