# Microsoft® Research Faculty Summit 2010

Microsoft® Research
Faculty Summit 2010

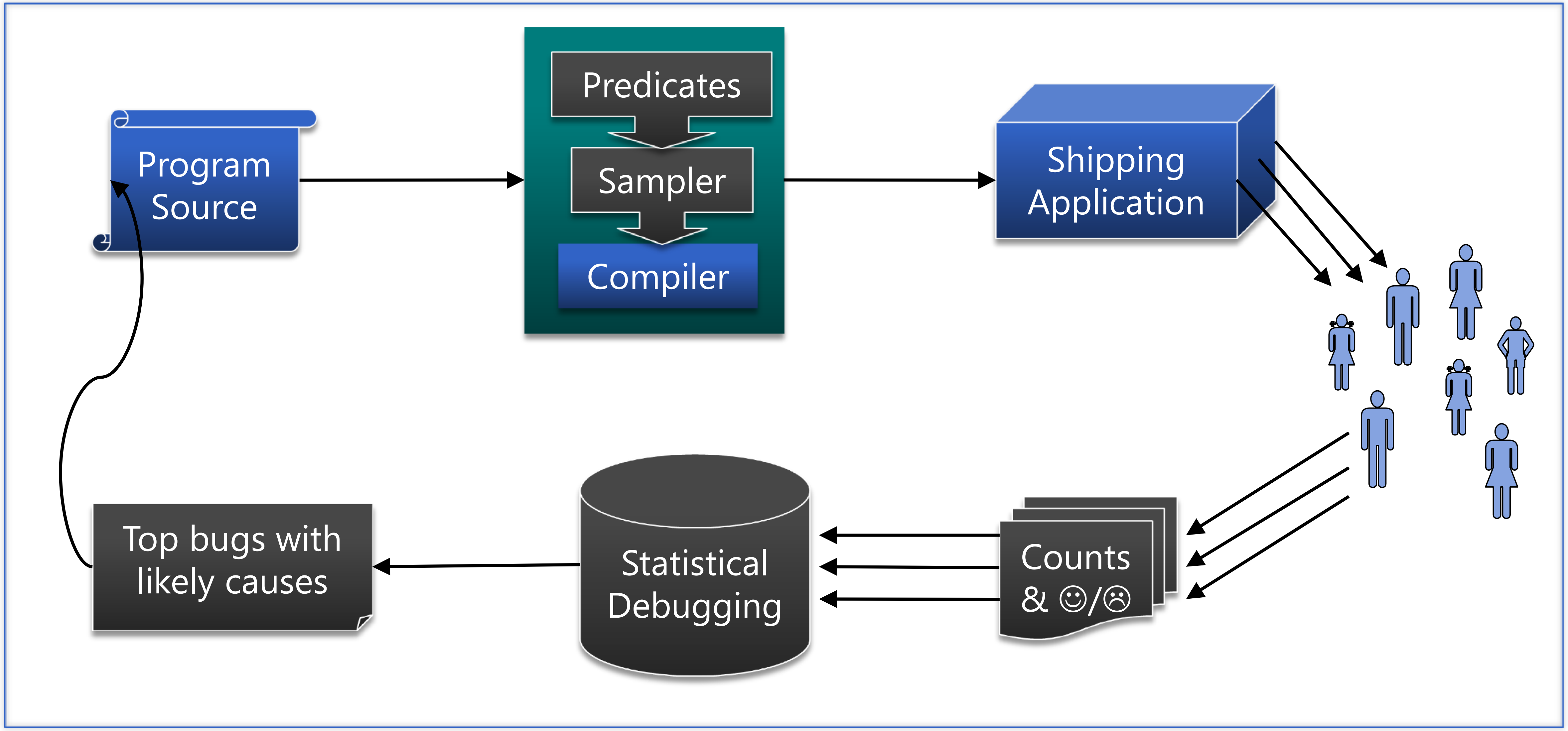
# Cooperative Debugging With One Million of Your Closest Friends

Ben Liblit,
University of Wisconsin–Madison

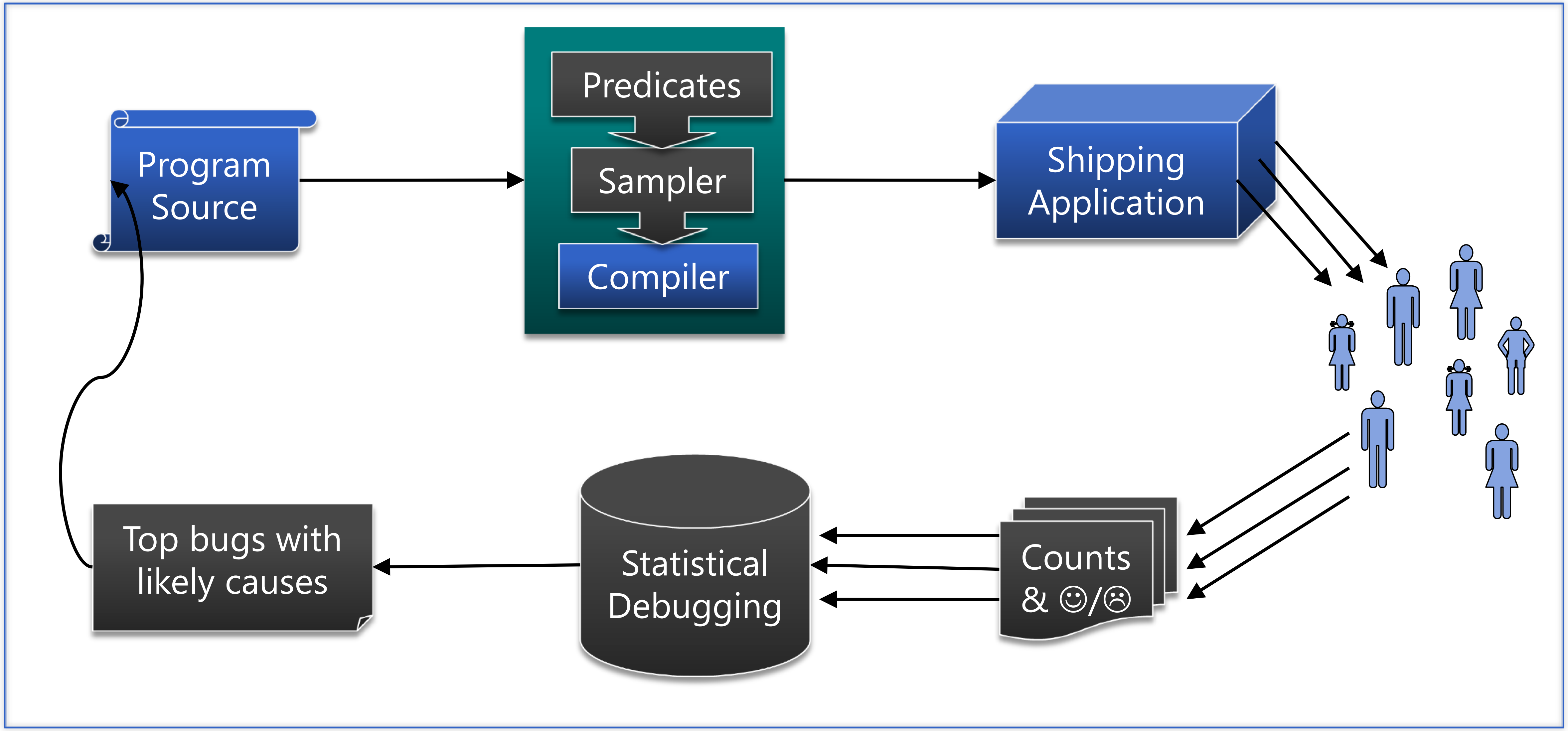
## Software in its Natural Environment

- Users vastly outnumber testers
  - 1 million runs of Microsoft Word per hour
- Real-world executions are most important
  - 1% of software errors cause 50% of user crashes
- Post-deployment bug hunting
  - Collect feedback data & mine for bug causes
  - Empirical and approximate, not exhaustive or absolute

# Cooperative Bug Isolation Architecture



# Predicate Injection: Guessing What's Interesting



## Returned Values Are Interesting

```
n = fprintf(…);
```

- Did you know that `fprintf()` returns a value?
- Do you know what the return value means?
- Do you remember to check it?

## Returned Value Predicate Counts

```
n = fprintf(…);
if (n < 0)  ++count[0];
if (n == 0) ++count[1];
if (n > 0)  ++count[2];
```

- Count how often n is negative/zero/positive
- Syntax yields instrumentation site
  - *Everywhere*, even if programmer forgot to check
- Site yields predicates on program behavior

## Many Other Behaviors of Interest

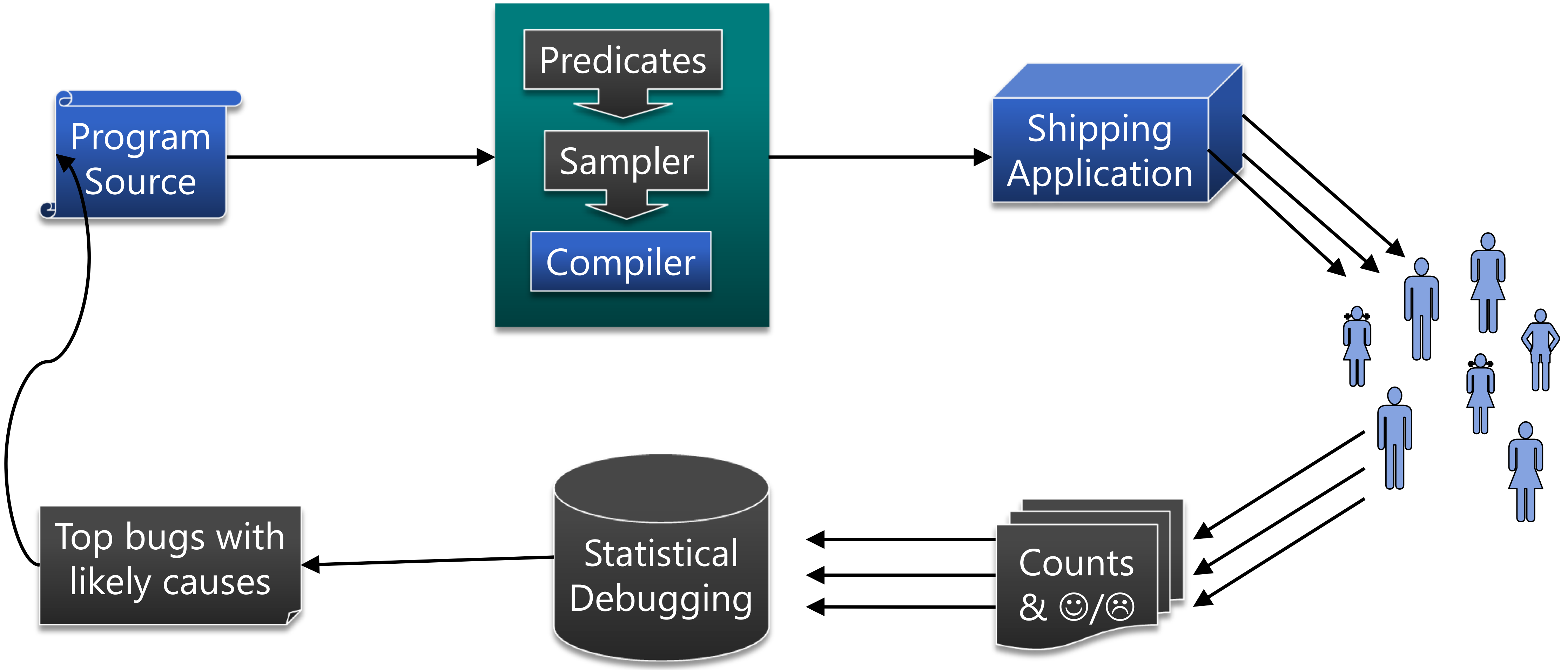
- Directions of branches
- `assert` statements
- Unusual floating point values
- Coverage of modules, functions, basic blocks, ...
- Reference counts: negative, zero, positive, invalid
- More ideas? Toss them all into the mix!

## Summarization and Reporting

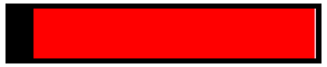
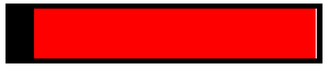
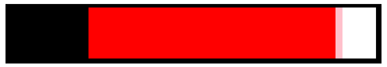
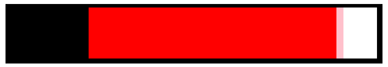
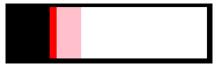
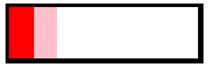
- Observation stream ➔ observation count
  - How *often* is each predicate observed true?
  - Removes time dimension, for good or ill
- Sparsely sampled for performance & privacy
  - Will not discuss this in detail today
- Feedback report is:

1. Vector of sampled predicate counters
2. Success/failure outcome label

# Playing the Numbers Game



# Example of Diagnostic Output

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

- 3 bug predictors from 156,476 initial predicates
- Compact visualization of many statistical measures
  - Big red bar ⇒ *strong* predictor of *common* bug!
- Each predicate identifies a distinct crashing bug

# Delta Latent Dirichlet Allocation (ΔLDA)

$$p(z_k = i \mid \mathbf{z}_{-k}, \mathbf{w}, \mathbf{o}) \propto \left( \frac{n^{i}_{-k,j_k} + \beta^{i}_{j_k}}{n^{i}_{-k,*} + \sum_{j'}^{W} \beta^{i}_{j'}} \right) \left( \frac{n^{d_k}_{-k,i} + \alpha^{o_k}_{i}}{n^{i}_{-k,*} + \sum_{i'}^{N_u + N_b} \alpha^{o_k}_{i'}} \right)$$

$$p(\mathbf{w}, \mathbf{z}) = p(\mathbf{w} \mid \mathbf{z})\, p(\mathbf{z})$$

$$p(\mathbf{w} \mid \mathbf{z}) = \prod_{i}^{N} \int p(\phi_i \mid \beta) \prod_{j}^{W} \phi_{ij}^{\,n^{i}_{j}} \, d\phi_i$$

$$p(\mathbf{z}) = \prod_{d}^{D} \int p(\phi_d \mid \alpha) \prod_{i}^{N} \phi_{di}^{\,n^{d}_{i}} \, d\phi_d$$

## Delta Latent Dirichlet Allocation (ΔLDA)

$$p(z_k = i \mid \mathbf{z}_{-k}, \mathbf{w}, \mathbf{o}) \propto \left( \frac{n^{i}_{-k,j_k} + \beta^{i}_{j_k}}{n^{i}_{-k,*} + \sum_{j'}^{W} \beta^{i}_{j'}} \right) \left( \frac{n^{d_k}_{-k,i} + \alpha^{o_k}_{i}}{n^{i}_{-k,*} + \sum_{i'}^{N_u + N_b} \alpha^{o_k}_{i'}} \right)$$

$$p(\mathbf{w}, \mathbf{z}) = p(\mathbf{w} \mid \mathbf{z})\, p(\mathbf{z})$$

$$p(\mathbf{w} \mid \mathbf{z}) = \prod_{i}^{N} \int p(\phi_i \mid \beta) \prod_{j}^{W} \phi_{ij}^{\,n^{i}_{j}} \, d\phi_i$$

$$p(\mathbf{z}) = \prod_{d}^{D} \int p(\phi_d \mid \alpha) \prod_{i}^{N} \phi_{di}^{\,n^{d}_{i}} \, d\phi_d$$

## Delta Latent Dirichlet Allocation (ΔLDA)

$$p(z_k = i \mid \mathbf{z}_{-k}, \mathbf{w}, \mathbf{o}) \propto \left( \frac{n^{i}_{-k,j_k} + \beta^{i}_{j_k}}{n^{i}_{-k,*} + \sum_{j'}^{W} \beta^{i}_{j'}} \right) \left( \frac{n^{d_k}_{-k,i} + \alpha_i^{o_k}}{n^{i}_{-k,*} + \sum_{i'}^{N_u + N_b} \alpha_{i'}^{o_k}} \right)$$

Let’s not talk about this right now, OK?

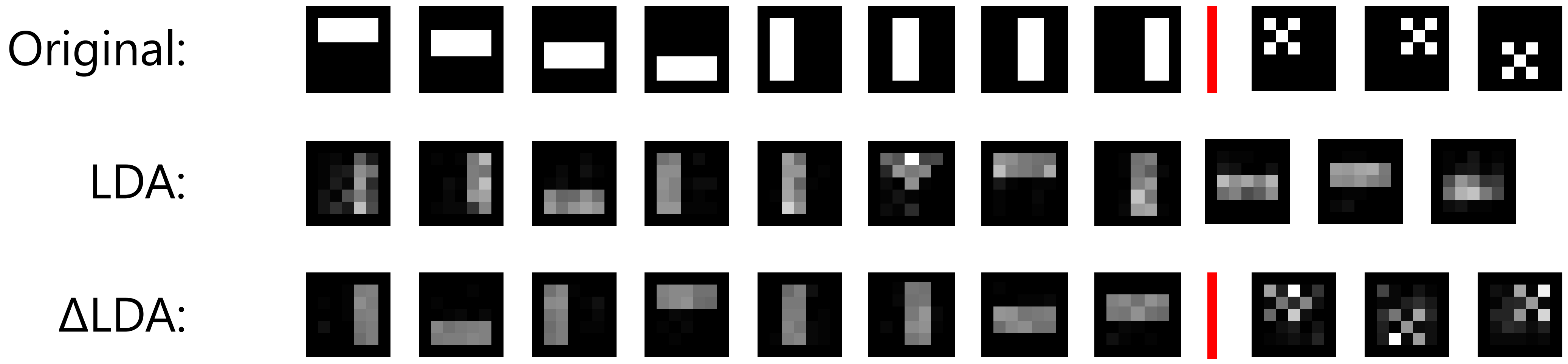
$$p(\mathbf{w}, \mathbf{z}) = p(\mathbf{w} \mid \mathbf{z})\, p(\mathbf{z})$$

$$p(\mathbf{w} \mid \mathbf{z}) = \prod_{i}^{N} \int p(\phi_i \mid \beta) \prod_{j}^{W} \phi_{ij}^{\,n_j^i}\, d\phi_i$$

$$p(\mathbf{z}) = \prod_{d}^{D} \int p(\phi_d \mid \alpha) \prod_{i}^{N} \phi_{di}^{\,n_i^d}\, d\phi_d$$

# ΔLDA in a Nutshell

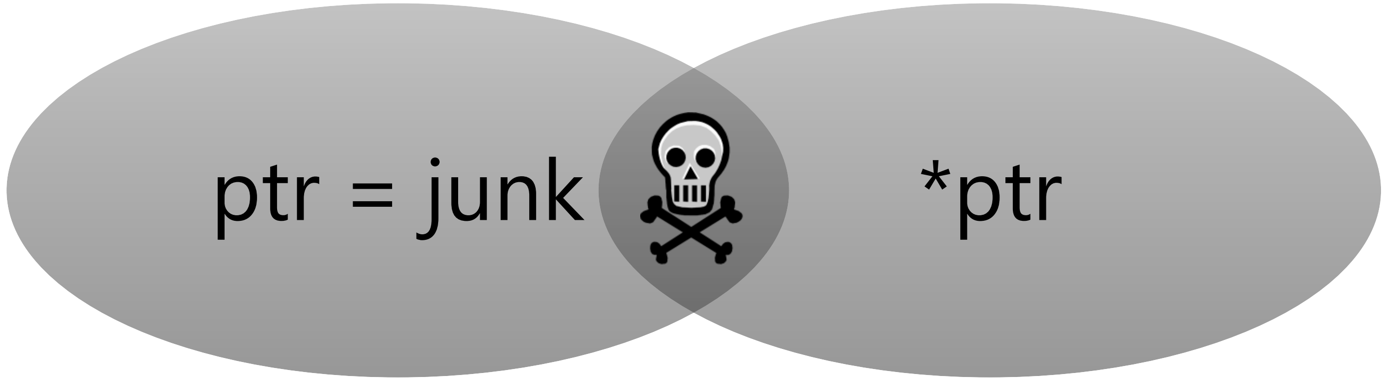
- Novel variant on text-document topic analysis
  - Program runs as "documents"
  - Program behaviors as "words," most of which are correct
- Failing runs draw upon reserved failure "topics"
  - Failure-topic "keywords" point to root causes of failures



## 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



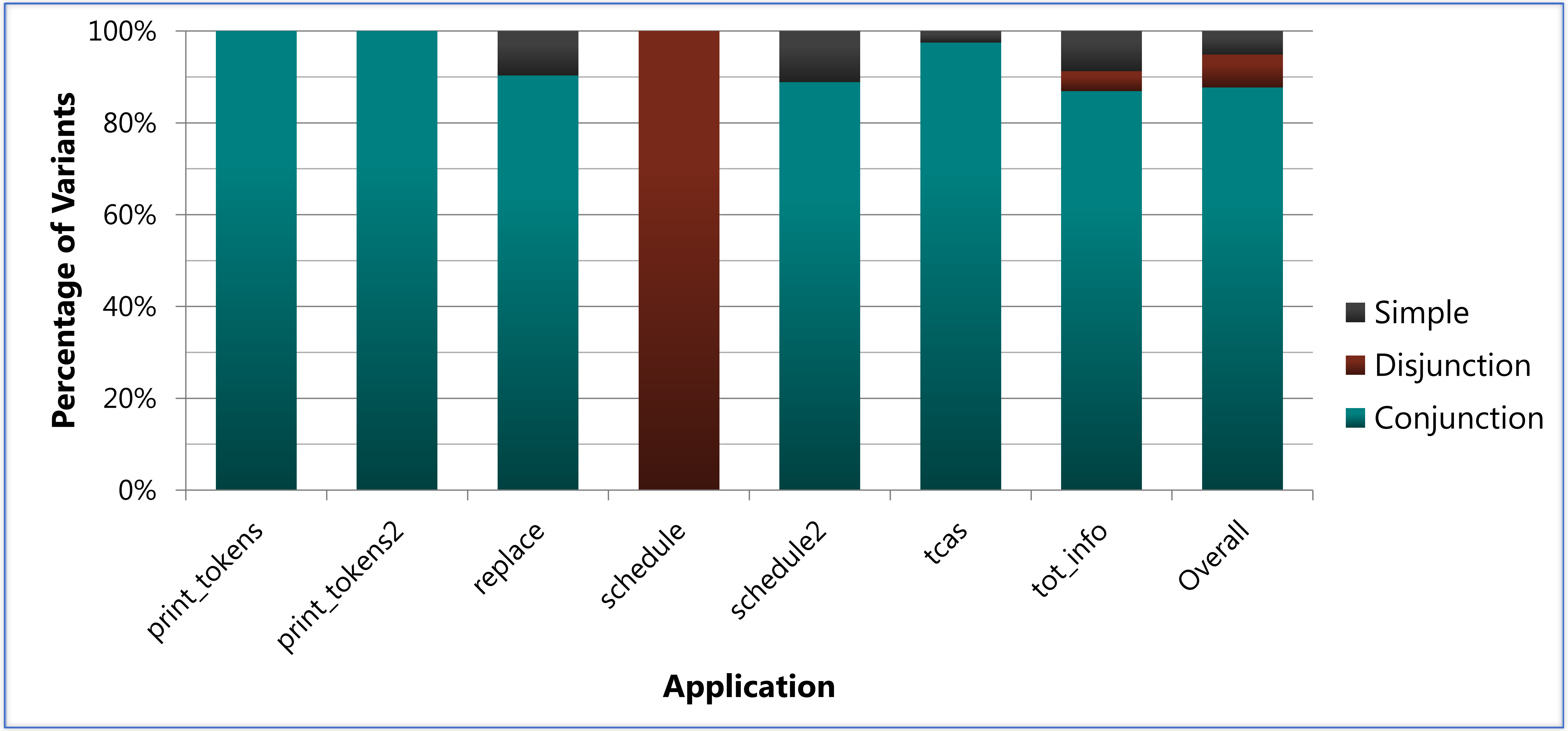
```
if (o + s > buf_size) return;
…
n->entries[i].data = malloc(s);
```

- Crash on later use of `n->entries[i].data`

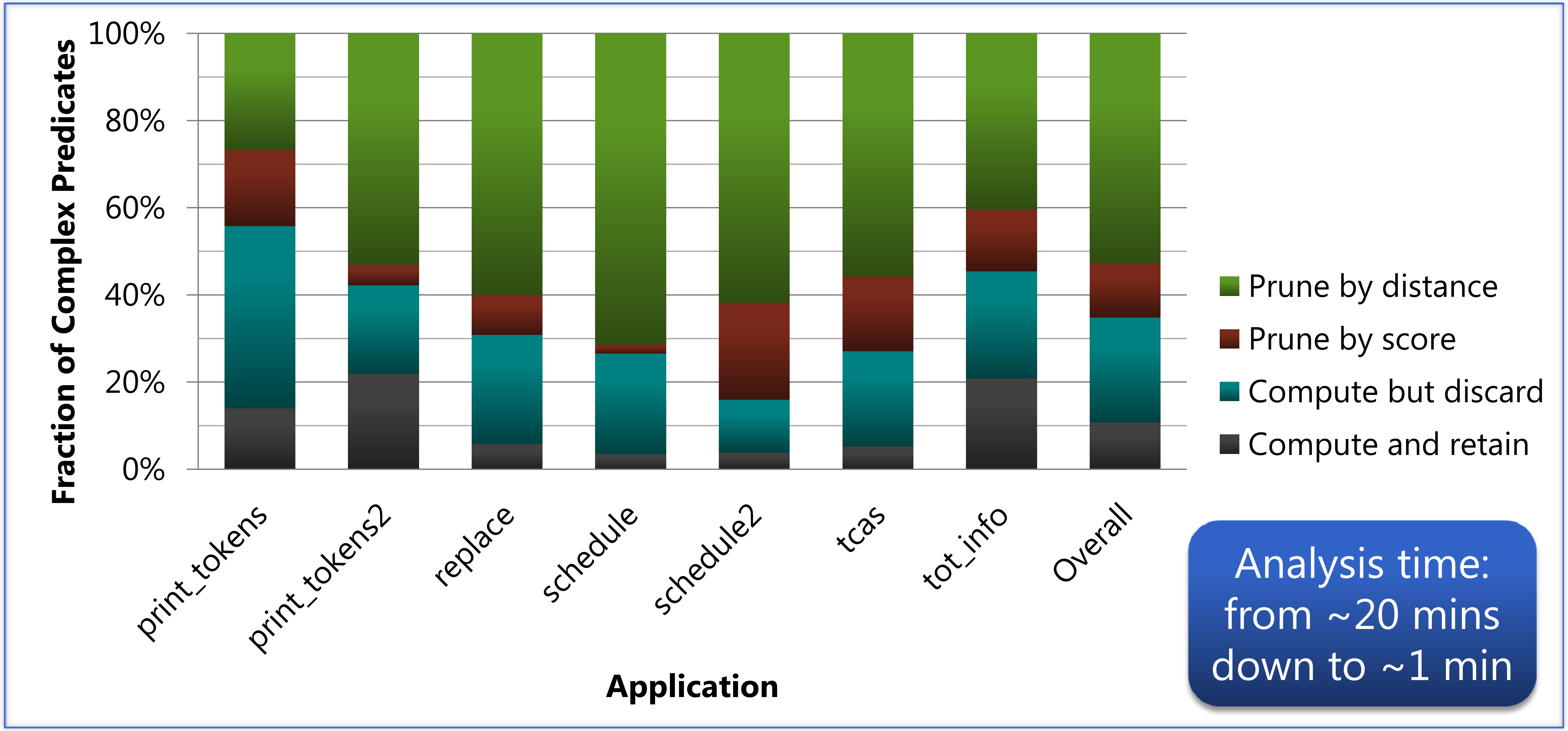
# Scalability Challenges and Solutions

- Too many compound pairings, even if inferred offline
  - Quadratic conjunctions & disjunctions of two predicates
  - 20 minutes for ~500 predicates, ~5,000 runs
  - Pruning optimizations needed!
- Discard if too far apart in program
  - Limit: 5% of program dependence graph
- Compute score upper bound and discard if too low
  - “Too low” = lower than constituent simple predicates
  - Reduces O(*runs*) to O(1) per complex predicate

# Evaluation: Kinds of Top Predicates

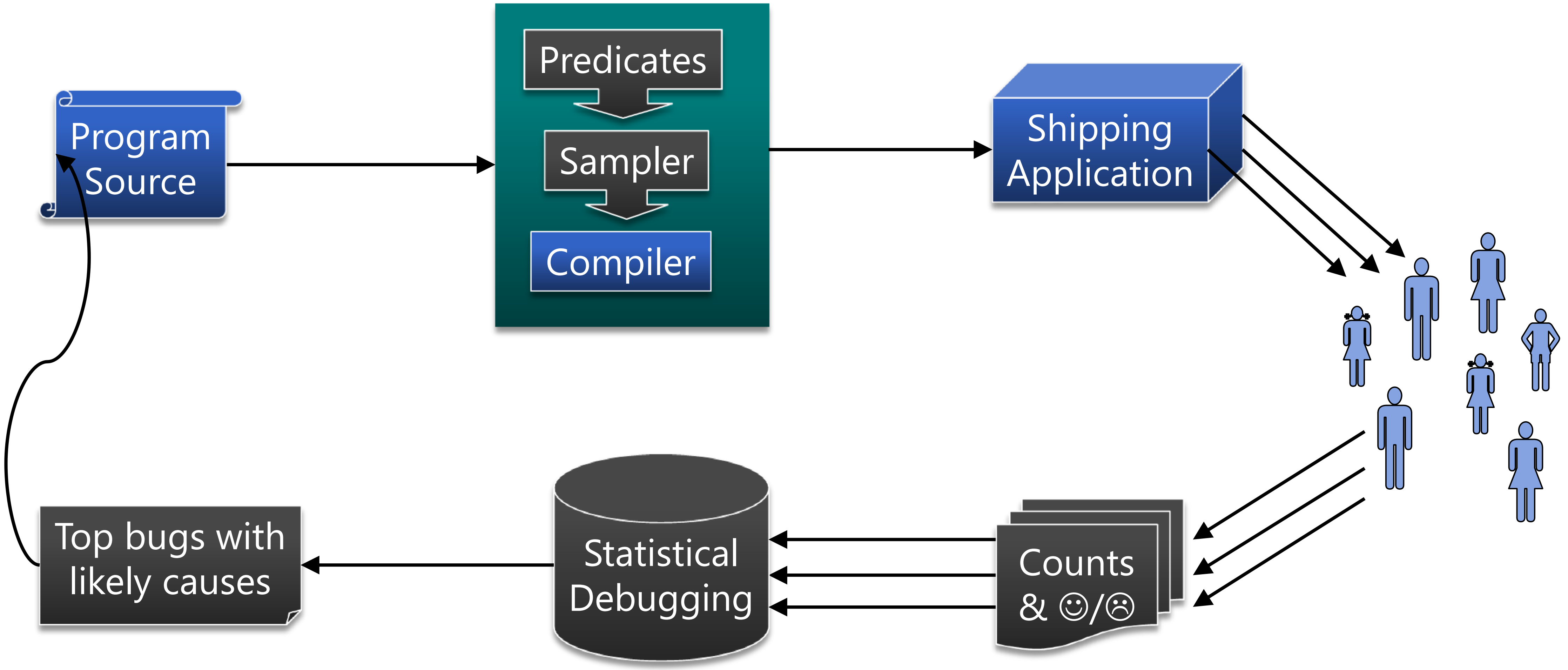


# Evaluation: Effectiveness of Pruning



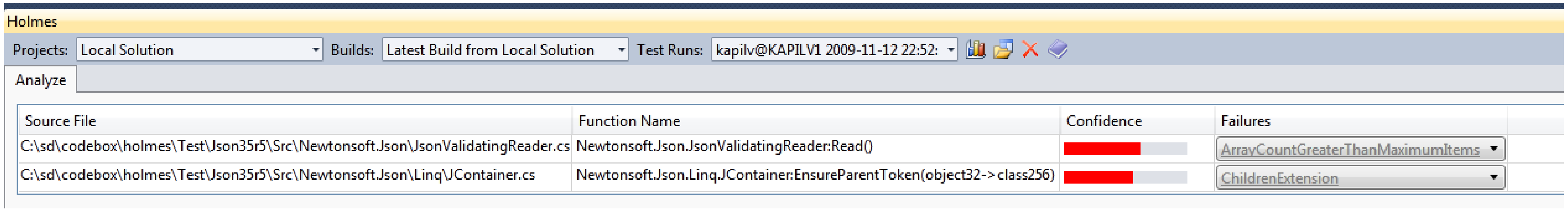
Analysis time: from ~20 mins down to ~1 min

# Putting Predictors in Context



## Visual Studio Integration

- Holmes: automated statistical debugging for .NET
  - Free download from Microsoft Research
- Run test suite & analyze results
- Visualize and explore buggy paths



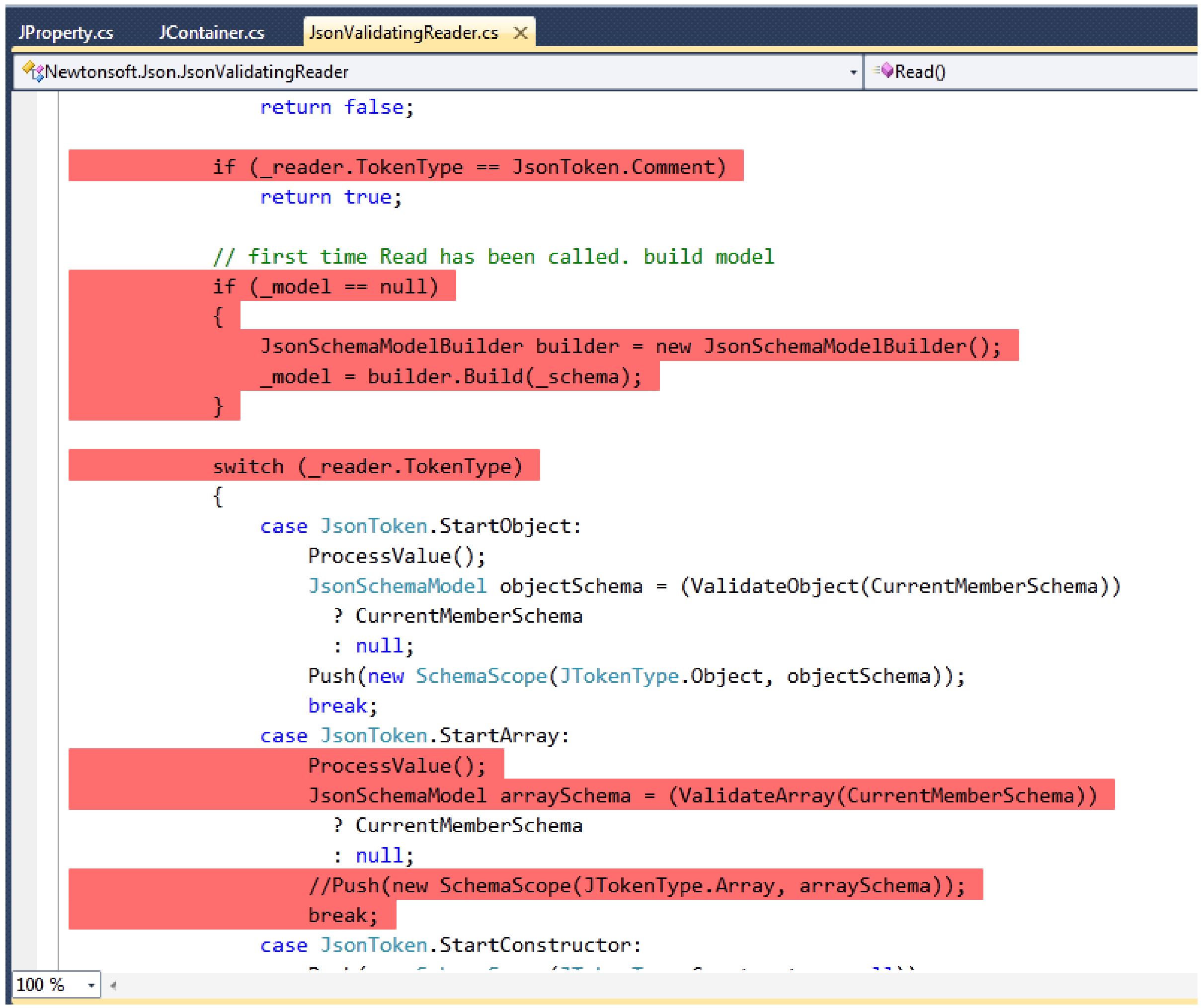
Holmes

Projects: Local Solution | Builds: Latest Build from Local Solution | Test Runs: kapilv@KAPILV1 2009-11-12 22:52:

Analyze

| Source File | Function Name | Confidence | Failures |
|---|---|---|---|
| C:\sd\codebox\holmes\Test\Json35r5\Src\Newtonsoft.Json\JsonValidatingReader.cs | Newtonsoft.Json.JsonValidatingReader:Read() | 🟥 | ArrayCountGreaterThanMaximumItems |
| C:\sd\codebox\holmes\Test\Json35r5\Src\Newtonsoft.Json\Linq\JContainer.cs | Newtonsoft.Json.Linq.JContainer:EnsureParentToken(object32->class256) | 🟥 | ChildrenExtension |

# Visual Studio Integration

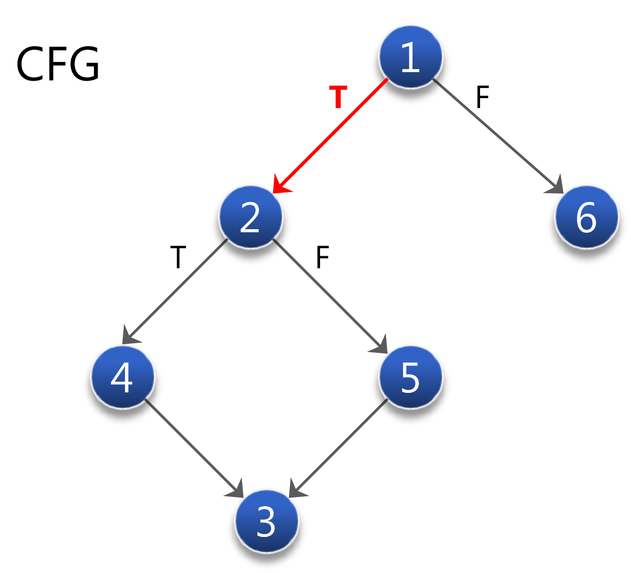


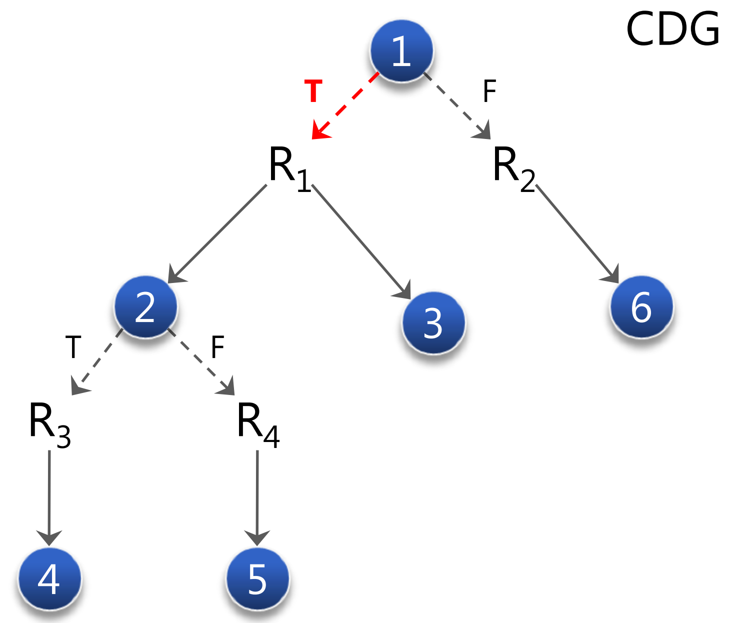
```
                return false;

            if (_reader.TokenType == JsonToken.Comment)
                return true;

            // first time Read has been called. build model
            if (_model == null)
            {
                JsonSchemaModelBuilder builder = new JsonSchemaModelBuilder();
                _model = builder.Build(_schema);
            }

            switch (_reader.TokenType)
            {
                case JsonToken.StartObject:
                    ProcessValue();
                    JsonSchemaModel objectSchema = (ValidateObject(CurrentMemberSchema))
                      ? CurrentMemberSchema
                      : null;
                    Push(new SchemaScope(JTokenType.Object, objectSchema));
                    break;
                case JsonToken.StartArray:
                    ProcessValue();
                    JsonSchemaModel arraySchema = (ValidateArray(CurrentMemberSchema))
                      ? CurrentMemberSchema
                      : null;
                    //Push(new SchemaScope(JTokenType.Array, arraySchema));
                    break;
                case JsonToken.StartConstructor:
```

## Killing Bugs: Shotgun or Rifle?

- Vast majority of code is correct
    - One experiment: 0.004% of predicates capture bugs
    - Saps performance, wastes user & developer resources
- Massive, static instrumentation is too blunt
    - Bug hunting with a shotgun
- Can we be bug snipers instead?
    - Highly selective instrumentation, guided by feedback
    - Binary rewriter to steer instrumentation through code
    - Iteratively follow trail of bugs back to root causes

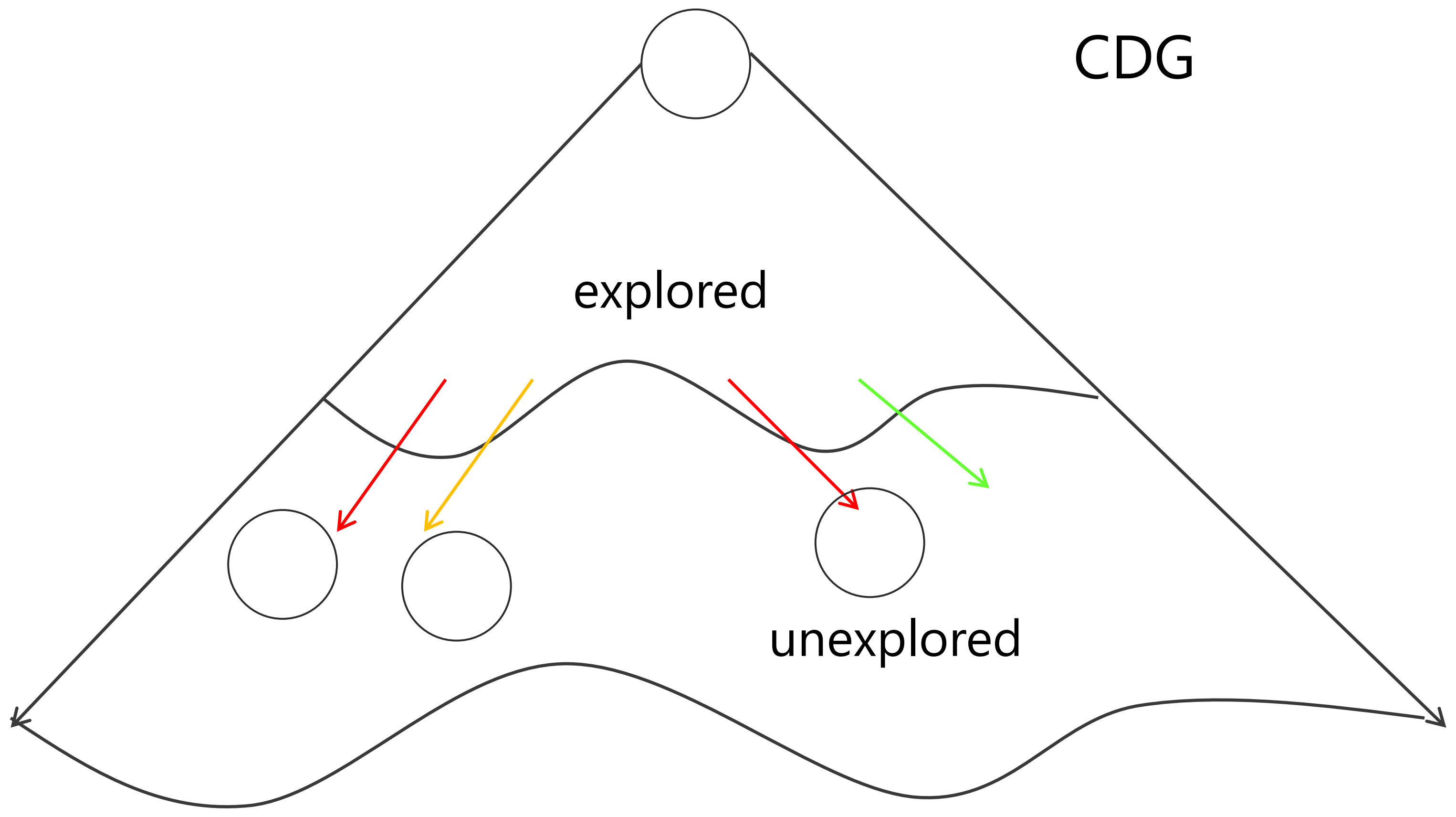
# Iterative Forward Search





- Blocks 2, 3, 4 and 5 dominated by faulty edge
- But only 2, 3 *immediately* dominated

# Broader View of CDG



# Impact on Selectivity and Performance

- Explore ~ 9% of large programs on average
  - bash, bc, ccrypt, exif, gcc
- Tens to hundreds of adaptation rounds

| Program | Sparsest Possible Static Sampling | Binary Instrumentation | |
|---|---|---|---|
| | | Complete | Adaptive |
| bc | 13% | 31% | 0.6% |
| gcc | 66% | 629% | 0.1% |
| gzip | 57% | 246% | 2% |
| Overall | 45% | 302% | 0.9% |

## Coming Soon: Concurrency Bugs

- Detect unusual thread interactions:
  - Function reentrance
  - Non-atomic access or update
- Deployable overhead via sparse sampling
- Rich statistical debugging data source
  - Catch races, atomicity violations, other bugs
  - Statistical methods not fooled by benign races

## Lessons Learned

- Can learn a lot from actual executions
  - Users are running buggy code anyway
  - We should capture some of that information
- Great potential in hybrid approaches
  - Dynamic: reality-driven debugging
  - Statistical: best-effort with uncertainty
  - Static: use program analysis to fill in the gaps

## Vision for Statistical Debugging

- 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!

## Join the Cause!

The Cooperative Bug Isolation Project

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

## Abstract

The resources available for testing and verifying software are always limited, and through sheer numbers an application’s user community will uncover many flaws not caught during development. The Cooperative Bug Isolation Project (CBI) marshals large user communities into a massive distributed debugging army to help programmers find and fix problems that appear after deployment. Dynamic instrumentation based on sparse random sampling provides our raw data; statistical machine learning techniques mine this data for critical bug predictors; static program analysis places bug predictors back in context of the program under study. We discuss CBI’s dynamic, statistical, and static views of post-deployment debugging and show how these different approaches join together to help improve software quality in an imperfect world.

## Bio

Ben Liblit is an Assistant Professor in the Computer Sciences Department of the University of Wisconsin–Madison.  Professor Liblit's research interests include programming languages and software engineering generally, with particular emphasis on combining machine learning with static and dynamic analysis for program understanding and debugging.

Professor Liblit worked as a professional software engineer for four years before beginning graduate study. His experience has inspired a research style that emphasizes practical, best-effort techniques that cope with the ugly complexities of real-world software development. Professor Liblit completed his Ph.D. in 2004 at UC Berkeley with advisor Alex Aiken, and received the 2005 ACM Doctoral Dissertation Award for his work on post-deployment statistical debugging.