Towards automatic debugging of concurrent programs

Elad Yom-Tov, Rachel Tzoref,
Shmuel Ur, Shlomo Hoory

Outline

- Introduction
  - The concurrent program debugging problem
  - Noise making
- Automatic debugging as a feature selection problem
- A batch solution to the problem
- Iterative Group Sampling

Why is debugging important?

- Ariane 5 flight 501
  - $500 million

- Mariner Climate Orbiter
  - $125 million

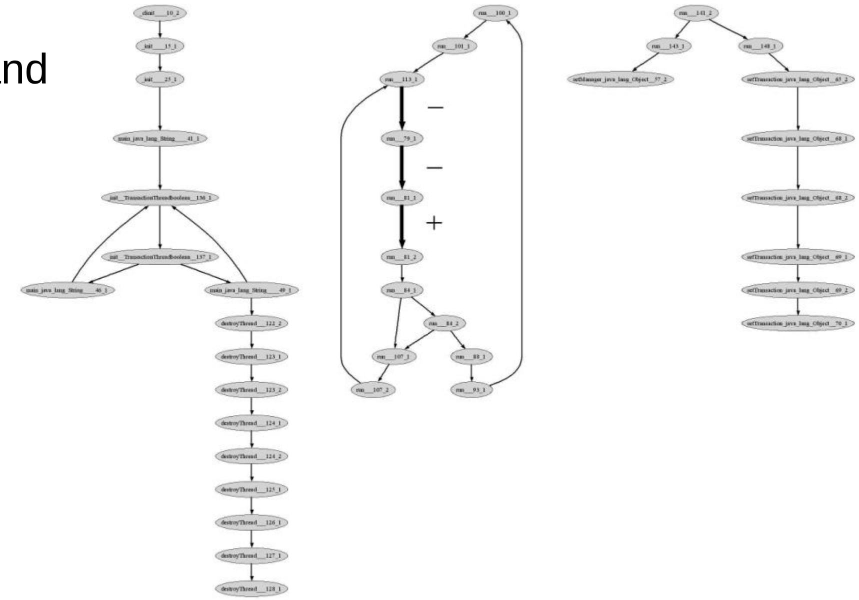
- “Software bugs, or errors, are so prevalent and so detrimental that they cost the US economy an estimated $59 billion annually, or about 0.6 percent of the gross domestic product” (NIST, 2002)

Concurrent programs

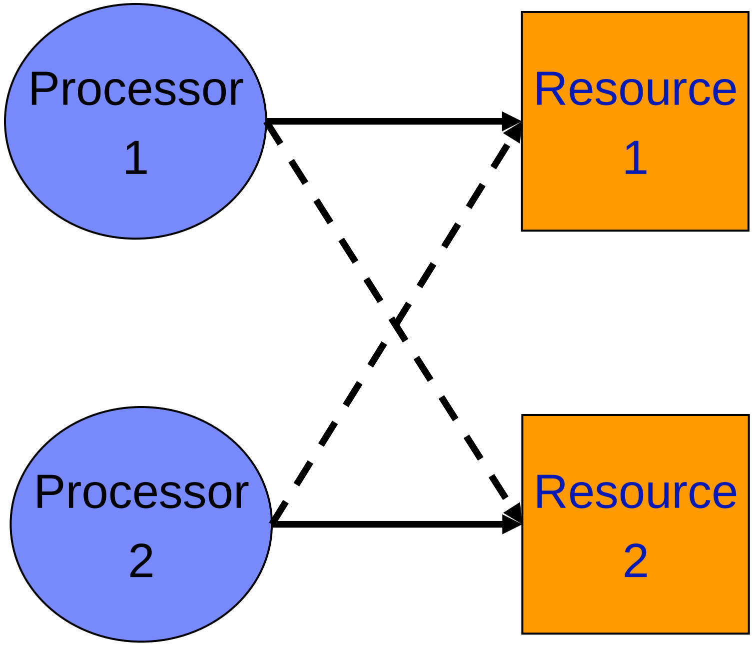
- Programs that are designed as collections of interacting computational processes that may be executed in parallel

- These programs are especially difficult to test and debug

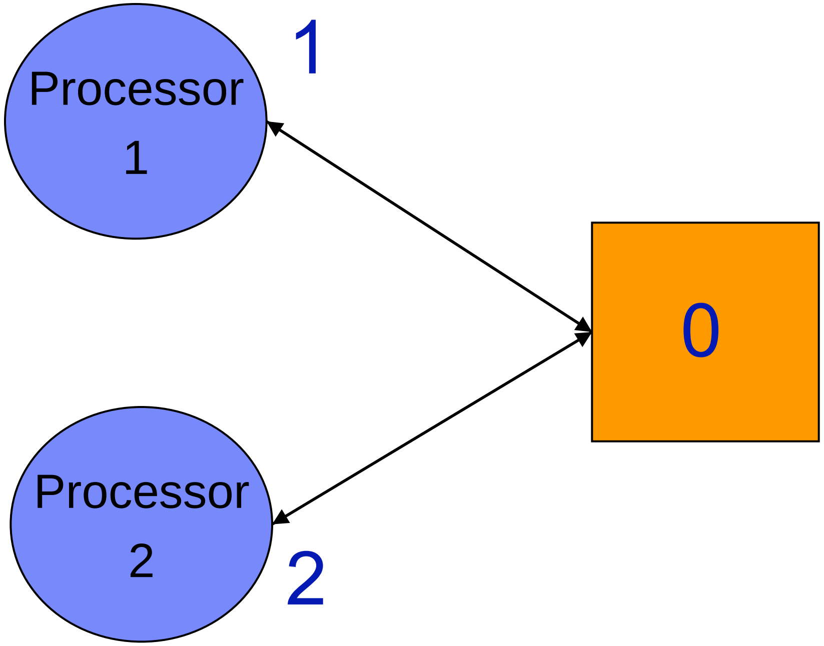
- Specific bugs include:
  - Deadlocks
  - Races



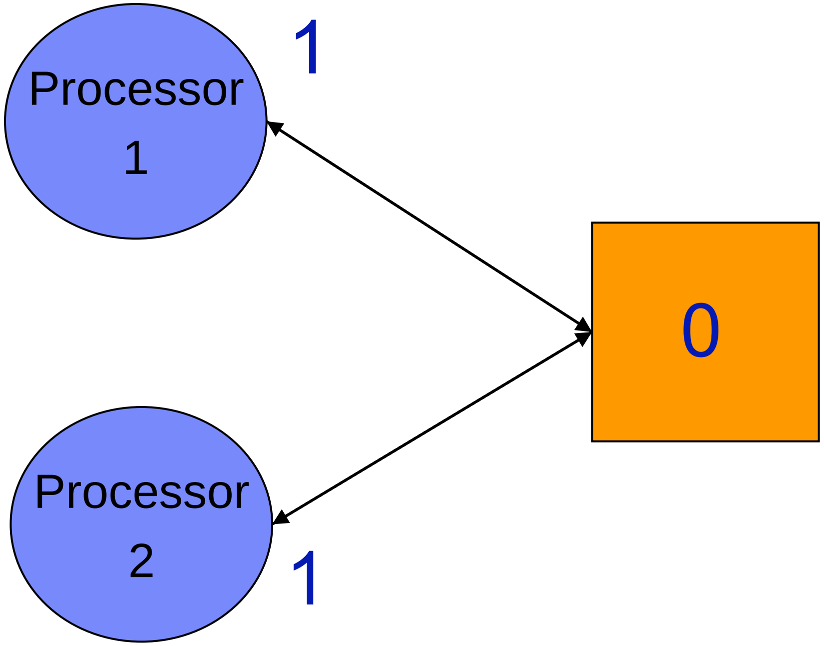
Deadlocks and races



Deadlocks and races

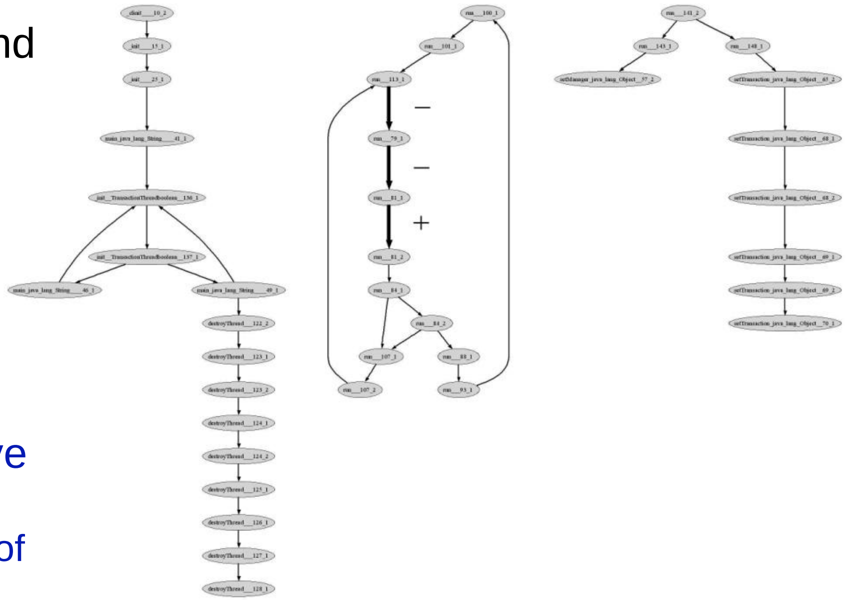


Deadlocks and races



Concurrent programs

- Programs that are designed as collections of interacting computational processes that may be executed in parallel
- These programs are especially difficult to test and debug
- Specific bugs include:
  - Deadlocks
  - Races
- Concurrency introduces **non-determinism**
  - Multiple executions of the same test may have different interleavings (and different results!)
    - An interleaving is the relative execution order of the program threads
  - Debugging changes interleaving, and can thus hide the bug



Eliciting bugs by adding timing noise to a program

- Find points which are “concurrently interesting”
  - Places whose relative interleaving may change the result of the program: access to shared variables, synch primitives
  - These points are called **instrumentation points**

- Modify the program by adding interleaving changing mechanisms
  - Usually sleep(), yield() but more advance mechanisms exist
  - Make sure that the interleaving change is random

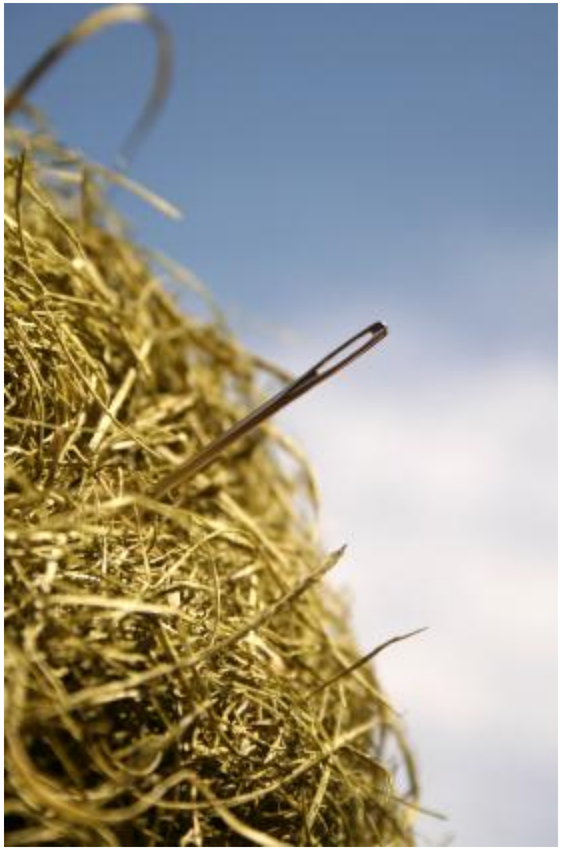
- Download http://www.alphaworks.ibm.com/tech/contest

Outline

- Introduction
  - The concurrent program debugging problem
  - Noise making
- Automatic debugging as a feature selection problem
- A batch solution to the problem
- Iterative Group Sampling

Debugging as a feature selection problem

- Think of each instrumentation point as a binary feature
- There are three kinds of instrumentation points:
  - Relevant
  - Irrelevant
  - Blocking
- Score instrumentation points:
  $P(i) = P(success|X_i)/P(!success|X_i)$
- The nodes with the highest score are usually related to location of failure



Possible ways to solve the problems

- Brute force search
- Delta debugging
- Random sampling of the space

Our approach to solving the problem

- Localize the bug:
  - Using offline sampling design
  - Using adaptive sampling
- Pinpoint the bug:
  - Identify specific problem points

Outline

- Introduction
  - The concurrent program debugging problem
  - Noise making
- Automatic debugging as a feature selection problem
- A batch solution to the problem
- Iterative Group Sampling

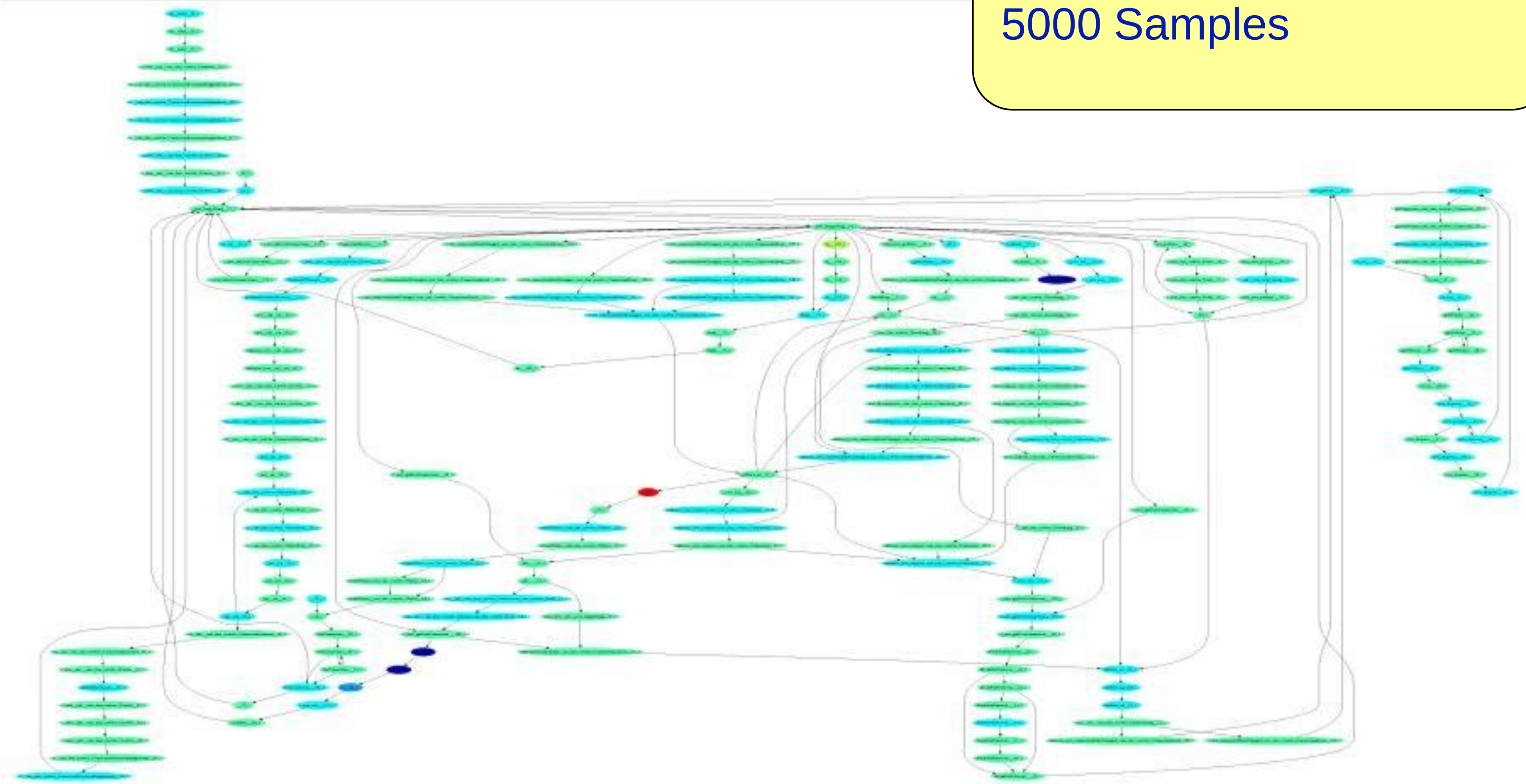
Stages to solution with offline sampling

1. Given a budget of ***N*** tests and ***M*** instrumentation points
2. Create an NxM sampling matrix
3. Execute N tests, each with a given instrumentation set
4. Score each instrumentation point and choose the points with the highest scores.

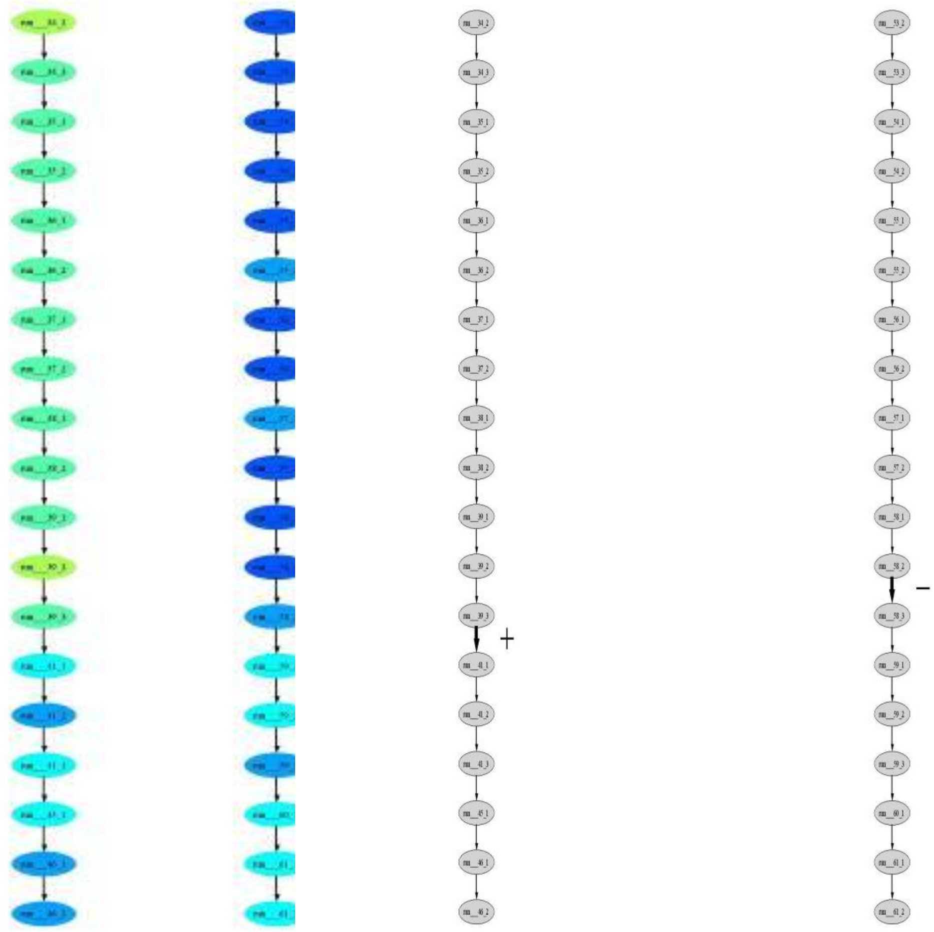
Crawler in WebSphere

314 Instrumentation points

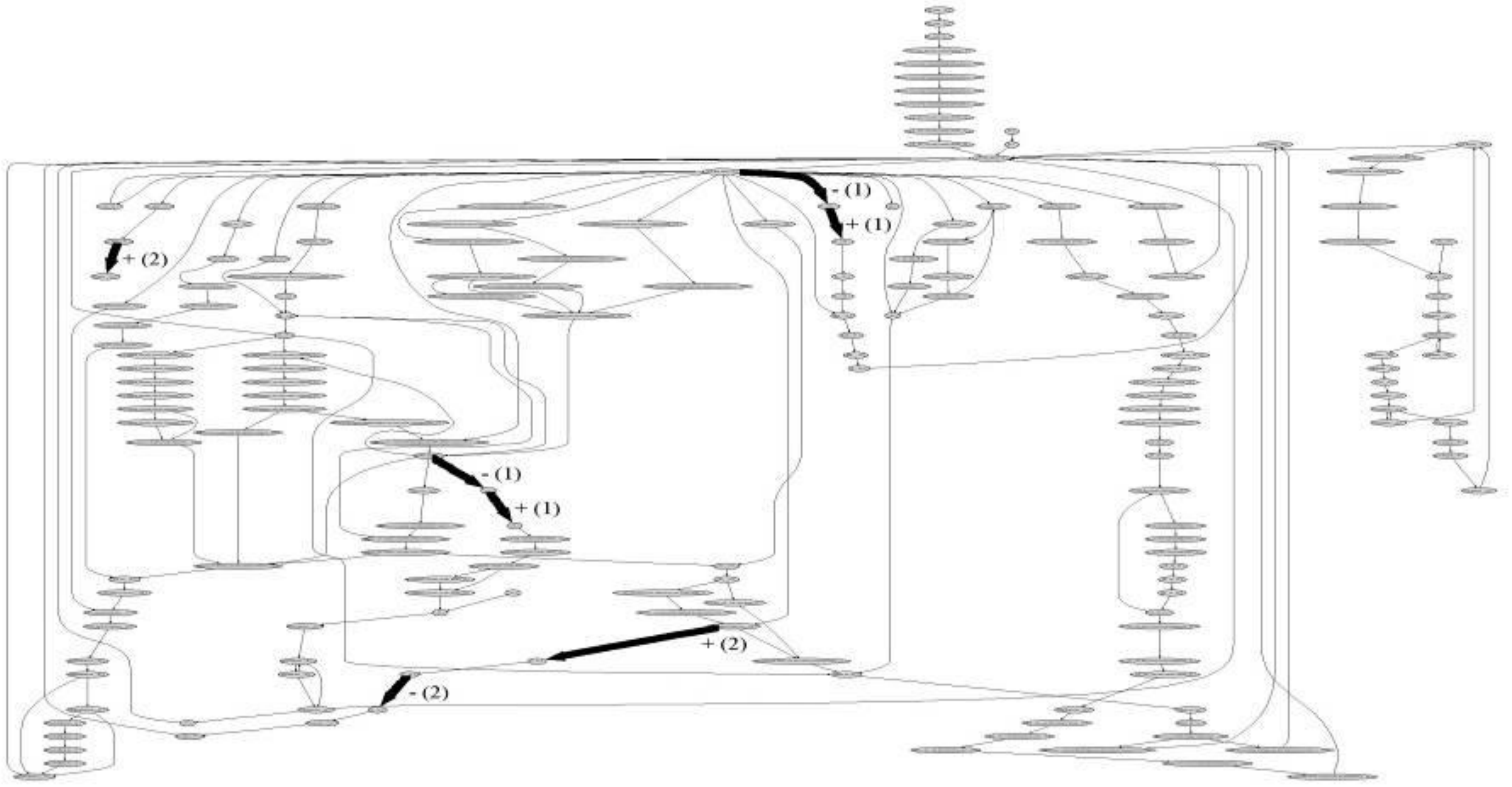
5000 Samples



This Does Not work When The Problem is Easy



The Derivative along the CFG of the WebSphere Crawler



Pros and cons of the offline sampling algorithm

- Pros:
  - We used about one order more samples than instrumentation points (Compared to $O(2^N)$ for brute force)
- Cons:
  - Batch sampling: Need to produce subsets of points in advance
  - Does not scale well: We used about one order more samples than instrumentation points
  - Batch localization: Need to compute derivative for the entire control flow graph of the program
  - **Does not utilize information gathered while sampling**

Outline

- Introduction
  - The concurrent program debugging problem
  - Noise making
- Automatic debugging as a feature selection problem
- A batch solution to the problem
- Iterative Group Sampling

Iterative Group Sampling (IGS)

- Divide the instrumentation points S randomly into L groups (L is a parameter specifying a low sampling dimension)
  - Every point i♍S belongs to one group in $G_1 \ldots G_L$
- While |S| >= |L|
  - Exhaustively sample all 2^L-2 binary vectors of length L
    - Point i is instrumented if the value for its corresponding group is true
    - If a bug is discovered, assign T(i) = 1, otherwise T(i)=0.
  - If the bug was discovered at least once during this iteration, identify groups that caused the bug to appear minimal number of times using T and discard all points within these groups from S.
- Return a set of relevant points S

- Note that the algorithm can be easily parallelized

Iterative Group Sampling – Running example

Assuming we have 16 instrumentation points

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

4,12 – relevant points for bug root cause

7,15 – blocking points that hide the bug

Randomly divide all points into 4 groups

Iterative Group Sampling – Running example

Now we run all possible group combinations

1 4 9 13	2 3 7 16	5 11 14 15	6 8 10 12

1 4 9 13 → Bug not exhibited

2 3 7 16 → Bug not exhibited

5 11 14 15 → Bug not exhibited

6 8 10 12 → Bug not exhibited

1 4 9 13 2 3 7 16 → Bug not exhibited

1 4 9 13 5 11 14 15 → Bug not exhibited

1 4 9 13 6 8 10 12 → Bug exhibited

…

1 4 9 13 2 3 7 16 5 11 14 15 → Bug not exhibited

Iterative Group Sampling – Running example

1 4 9 13

6 8 10 12

Divide remaining points into 4 groups

Run all possible group combinations

Discard all groups that exhibit the bug minimal number of times

Continue until reaching one point per group

Analysis for the probability of success

The algorithm will converge if in N_{Iter} iterations with probability θ:

$$(1-\Pr_{Good})^{N_{Iter}} \leq \theta$$

The probability to reach a good configuration can be computed as follows:

$$Pr_{Good}(K_r, K_b, L) = \sum_{i=1}^{\min(K_r, L)} P_r(K_r, L, i) \cdot P_b(K_b, L, i)$$

$$P_b(K_b, L, i) = \left(\frac{L-i}{L}\right)^{K_b}$$

$$P_r(K_r, L, i) = \frac{\binom{L}{i}}{L^{K_r}} \sum_{j=0}^{i-1} \binom{i}{j} (i-j)^{K_r} (-1)^j$$

Selecting the number of groups (worst case scenario)

It follows (with a little math) that in the total number of tests is:

$$T_{Total}(K_r, K_b, L, \theta) = \frac{\log(\theta)}{\log(1 - Pr_{Good})} \cdot (2^L - 2)$$

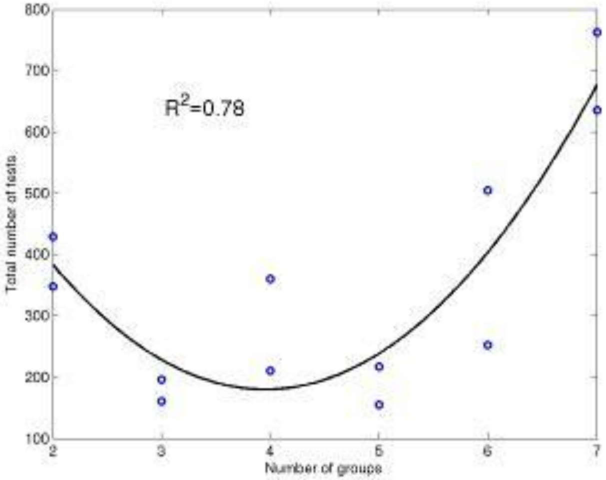
The probability for a bad partition is maximized when K_r=1. In this case:

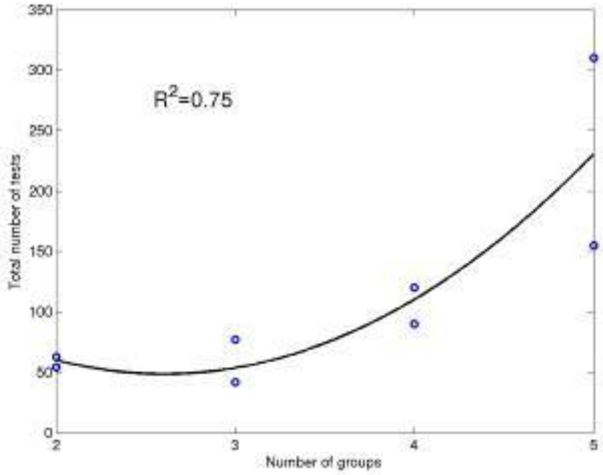
$$T_{Total,\ worse\ case} \geq \frac{\log(\theta)}{\log\left(1 - \left(\frac{L-1}{L}\right)^{K_b}\right)} \cdot (2^L - 2)$$

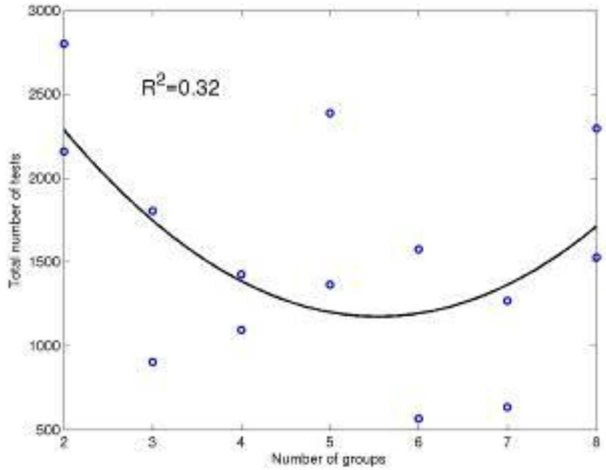
It follows that the minimal value for T_{Total} is achieved when:

$$L \approx \sqrt{K_b / \log(2)}.$$

Total number of tests as function of the number of groups







Results

Program number	Average precision Percent of relevant points		Average detection Was there a relevant point?	
	IGS	RELIEF	IGS	RELIEF
1	0.54	0.04	1.00	0.16
2	0.46	0.12	1.00	0.46
3	0.18	0.21	0.77	0.62

- Number of runs went down from 5000 to 70

Speeding up the IGS algorithm

Main idea: Save only groups with significant elicitation of bugs.

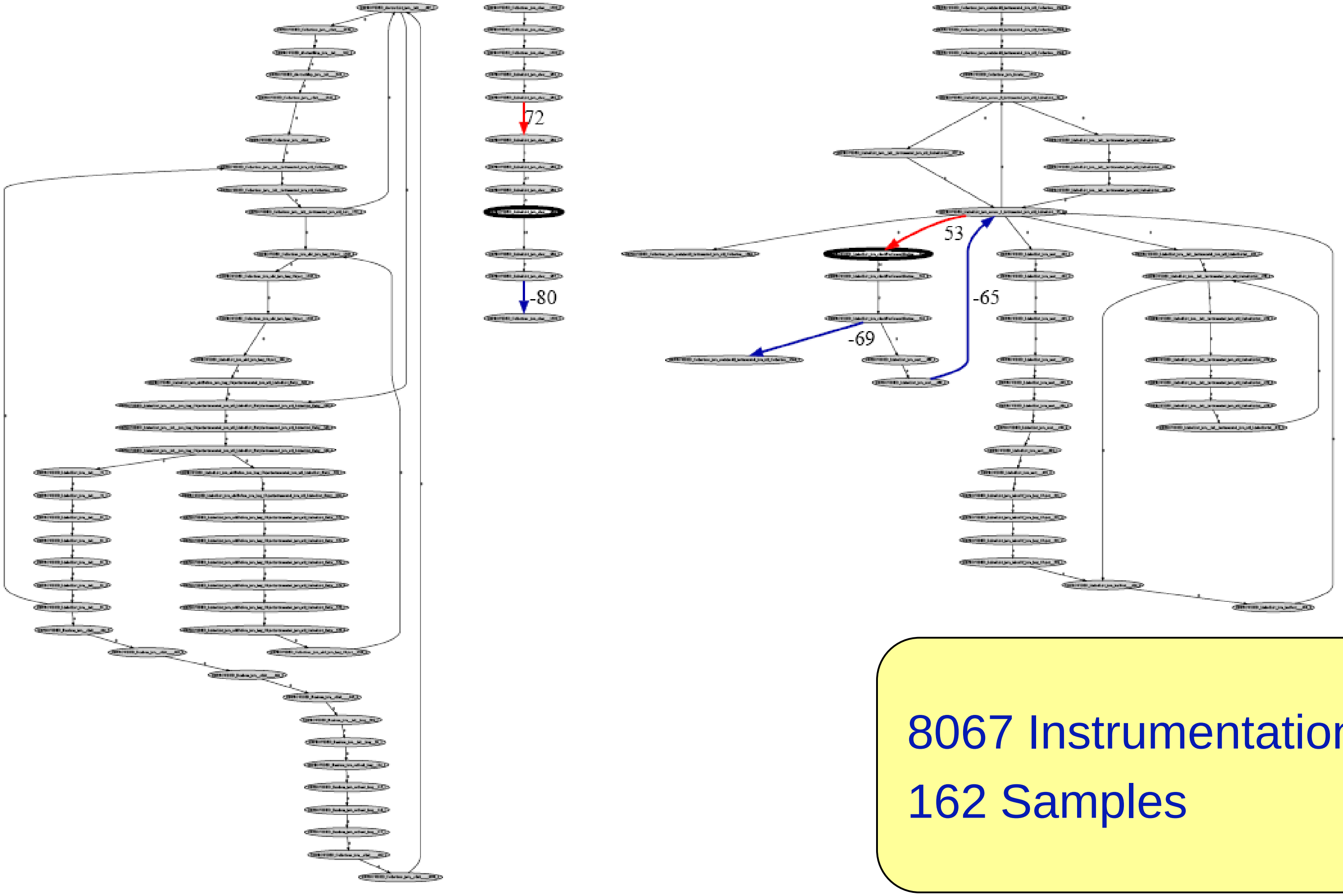
Method: χ^2 test based on a Normal approximation to the Binomial distribution

$$S = \frac{a - c}{\sqrt{(a + c) * (2 * n - a - c)/(2 * n)}}$$

Calculating the derivative score

- In the batch algorithm we showed that it is better to observe the derivative score along the control flow graph
- In the IGS algorithm calculating the derivative score is problematic
  - There is no score per point for all instrumentation points
- Solution: Calculate local derivative score on the control flow graph around the points identified by IGS

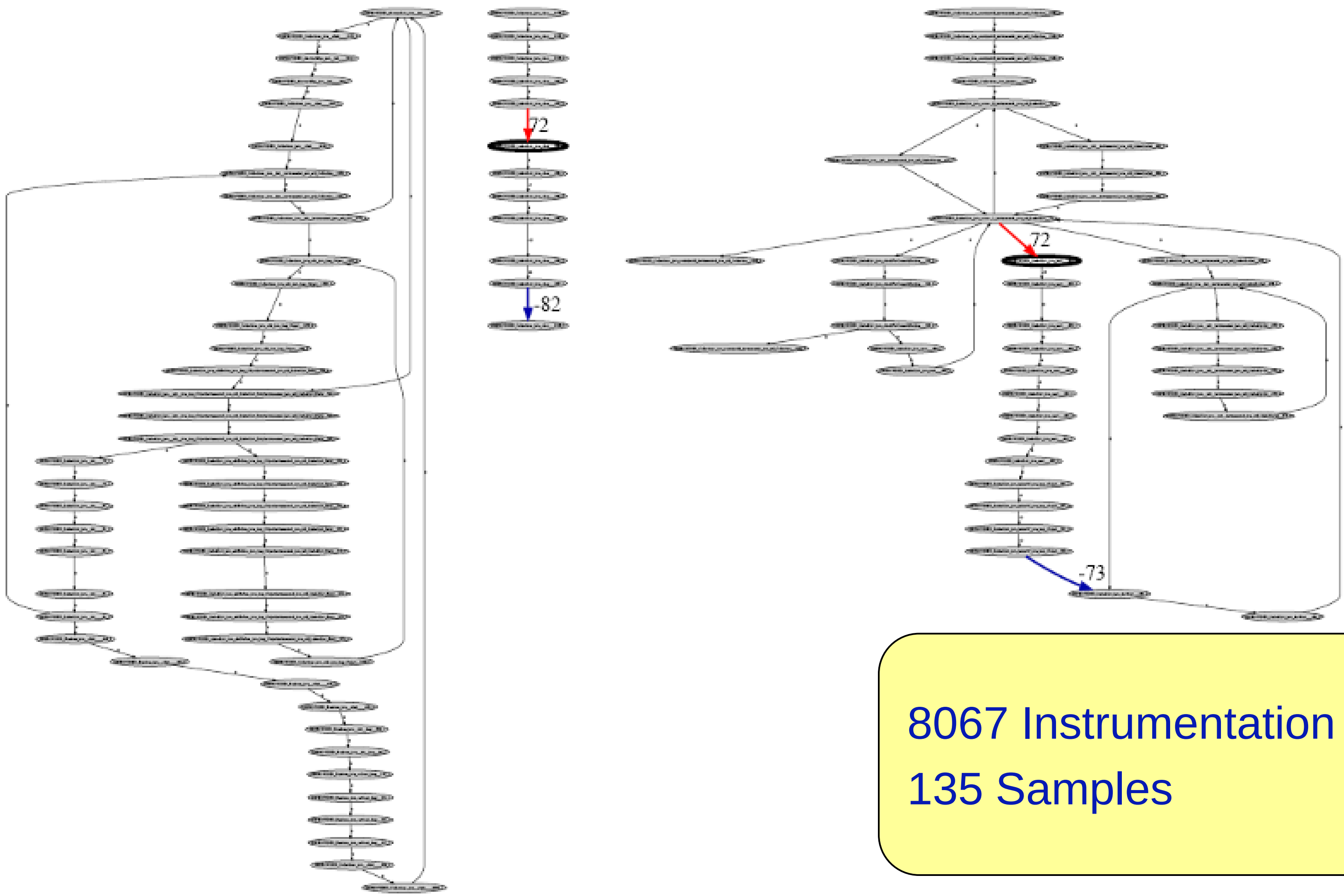
Java 1.4 Collection Library - Element exception bug



8067 Instrumentation points

162 Samples

Java 1.4 Collection Library - Infinite loop bug

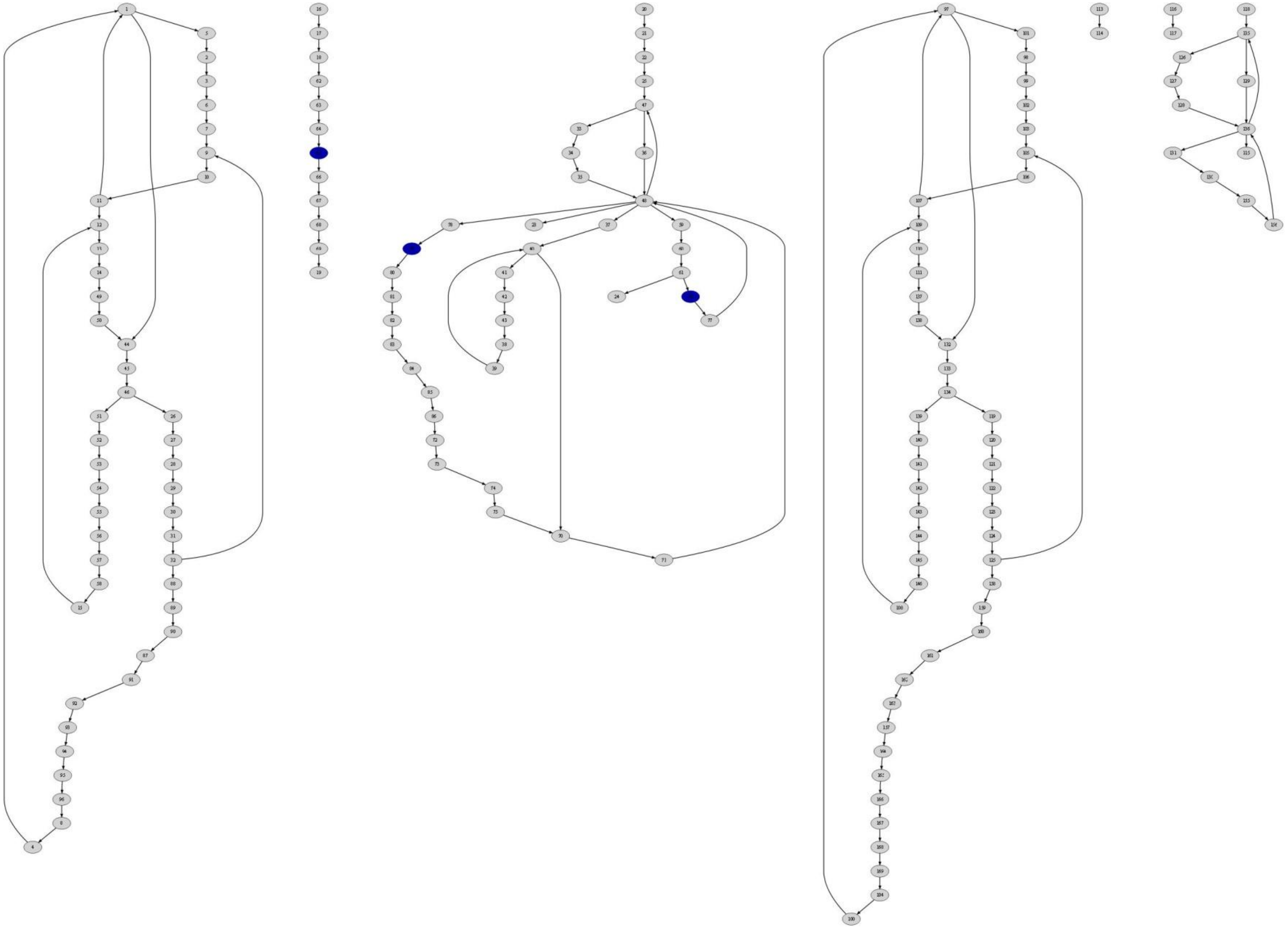


8067 Instrumentation points

135 Samples

Summary

- We presented an efficient automatic method for debugging concurrent programs.
- Using instrumentation of points within a program and noise injection it is possible to elicit bugs and pinpoint locations in the program code that are strongly suggestive of the source of the bugs.
- We demonstrated that finding an indicative minimal set can be posed as a problem of active feature selection and suggested the use of the IGS algorithm for solving this problem.
- We have recently applied this algorithm to software with 218,000 instrumentation points, and pinpointed a bug in under 7 hours!



धन्यवाद
Hindi

多謝
Traditional Chinese

ขอบคุณ
Thai

Спасибо
Russian

Gracias
Spanish

多谢
Simplified Chinese

Thank You
English

KIITOS
Danish

شكرا
Arabic

תודה
Hebrew (Toda)

Danke
German

Grazie
Italian

Merci
French

நன்றி
Tamil

ありがとうございました
Japanese

감사합니다
Korean

Obrigado
Portuguese