Static and Dynamic Verification of Concurrent Programs

LAB Session

Aarti Gupta
Systems Analysis & Verification
NEC Labs America, Princeton, USA

Third Summer School on Formal Techniques
May 20 – 24, 2013

NEC Laboratories America
Relentless passion for innovation

www.nec-labs.com

Lab Session

- **CalFuzzer**
  - http://srl.cs.berkeley.edu/~ksen/calfuzzer/

Other publicly available tools

- **Related tool: Thrille for UPC (Unified Parallel C)**
  - http://upc.lbl.gov/thrille
- **Threader**
  - http://www.model.in.tum.de/~popeea/research/threader.html
- **Model Checking: Java PathFinder**
  - http://ti.arc.nasa.gov/tech/rse/vandv/jpf/
- **Systematic testing: CHESS**
  - http://research.microsoft.com/en-us/projects/chess/
- **Static Analysis: Chord**
  - http://pag.gatech.edu/chord
- **Dynamic Analysis: RoadRunner**
  - http://dept.cs.williams.edu/~freund/rr/
- **Intel Thread Checker**
  - http://software.intel.com/en-us/articles/intel-thread-checker-documentation

CalFuzzer Tool

- **Thanks**
  - Prof. Koushik Sen (UC Berkeley)
  - Pallavi Joshi (UC Berkeley, now at NEC Labs)
  - Chang-Seo Park (UC Berkeley, now at Google)

- **Highlights**
  - Incorporates many techniques we discussed
    - Static/dynamic analysis: to find "potential violation"
    - Testing: to find real violation, based on above
  - Extensible – add your own analysis and checker!

Getting started …

- **Download CalFuzzer 2.0**

  http://srl.cs.berkeley.edu/~ksen/calfuzzer/

- **Build using ant**

```
tar zxvf calfuzzer2.tar.gz
cd calfuzzer
ant
ant -f run.xml racefuzzer
ant -f run.xml deadlockfuzzer
```

- **Small examples in:** `test/benchmarks/testcases`

- **To build and run an individual example (already included in run.xml)**

```
ant -f run.xml test_race1
```

- **To try a new example, add build commands to run.xml (similar to above)**
  - See example from Gidon Ernst (ConcurrentStack.java, on SSFT13 website)

Presentation of CalFuzzer Tool at CAV 2009

An Extensible Active Testing Framework for Concurrent Programs

Pallavi Joshi *
Mayur Naik‡
Chang-Seo Park★
Koushik Sen★

★ Par Lab, EECS, UC Berkeley ‡ Intel Research

Goal

- Build a tool to test and debug concurrent programs
  - More Practical: That works for large programs
  - Efficient
  - No false alarms
  - Finds many bugs quickly
  - Reproducible

Related Work: Concurrent Program Analysis

- Static program analysis (e.g., Engler et al.; Naik et al.)
  - 👍 Examines all possible program behavior
  - 👎 Often reports many false positives
- Type systems (e.g., Boyapati et al., Flanagan and Qadeer)
  - 👎 Annotation burden often significant
- Model checking (e.g., SPIN, Verisoft, Java Pathfinder)
  - 👎 Does not currently scale beyond few KLOC
  - 👎 Not "directed" towards finding bugs
- Dynamic program analysis (e.g. Eraser, Atomizer)
  - 👍 Usually reports lesser false positives
  - 👎 Has false negatives
- Testing
  - 👍 Scales to large programs and no false positives
  - 👎 False negatives and poor coverage

Observation

- Static and dynamic program analyses have false positives
- Testing is simple
  - No false positives
  - But, may miss subtle thread schedules that result in concurrency bugs

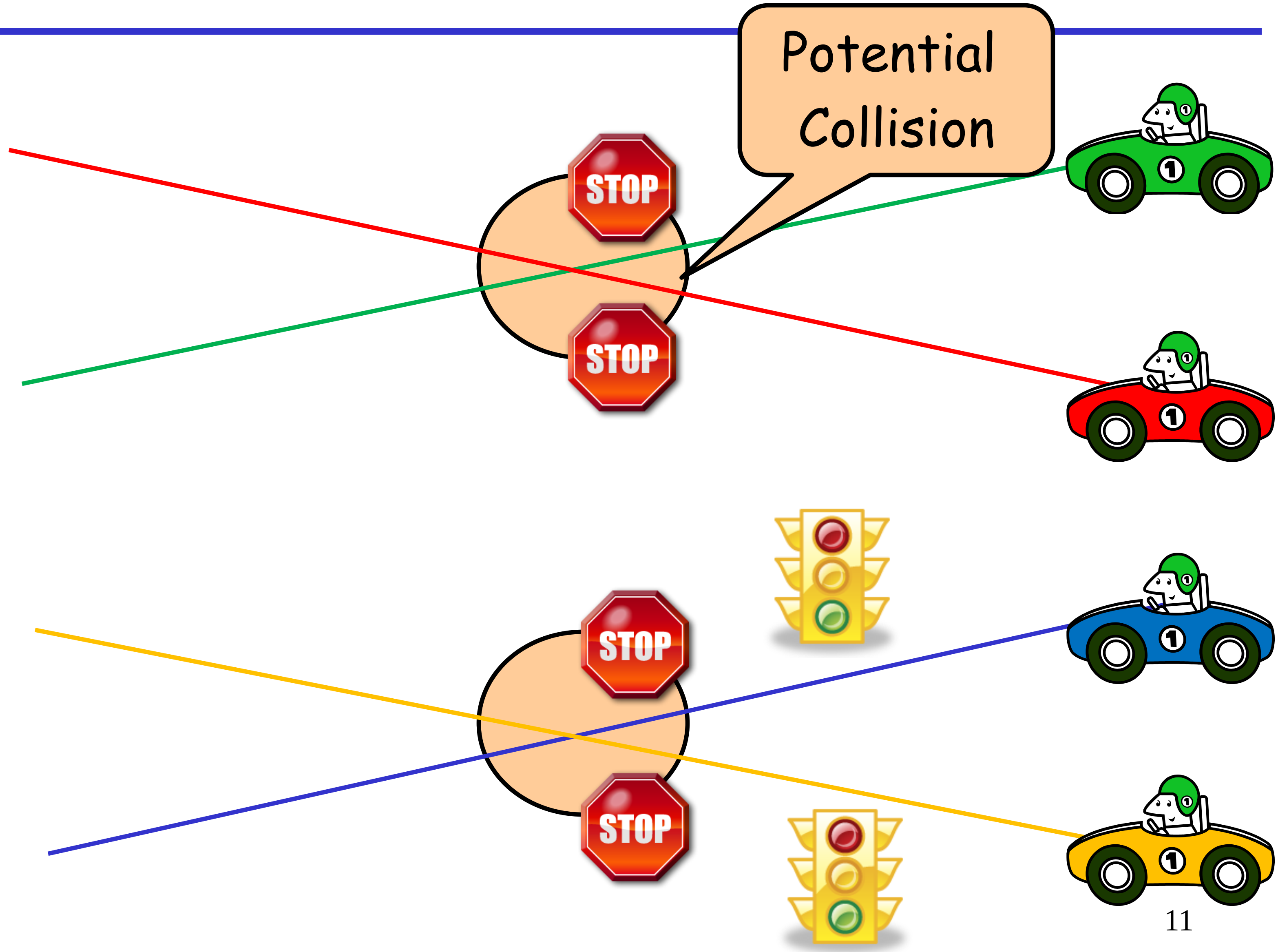
Observation

- Static and dynamic program analyses have false positives
- Testing is simple
  - No false positives
  - But, may miss subtle thread schedules that result in concurrency bugs
- Can we leverage program analysis to make testing quickly find real concurrency bugs?

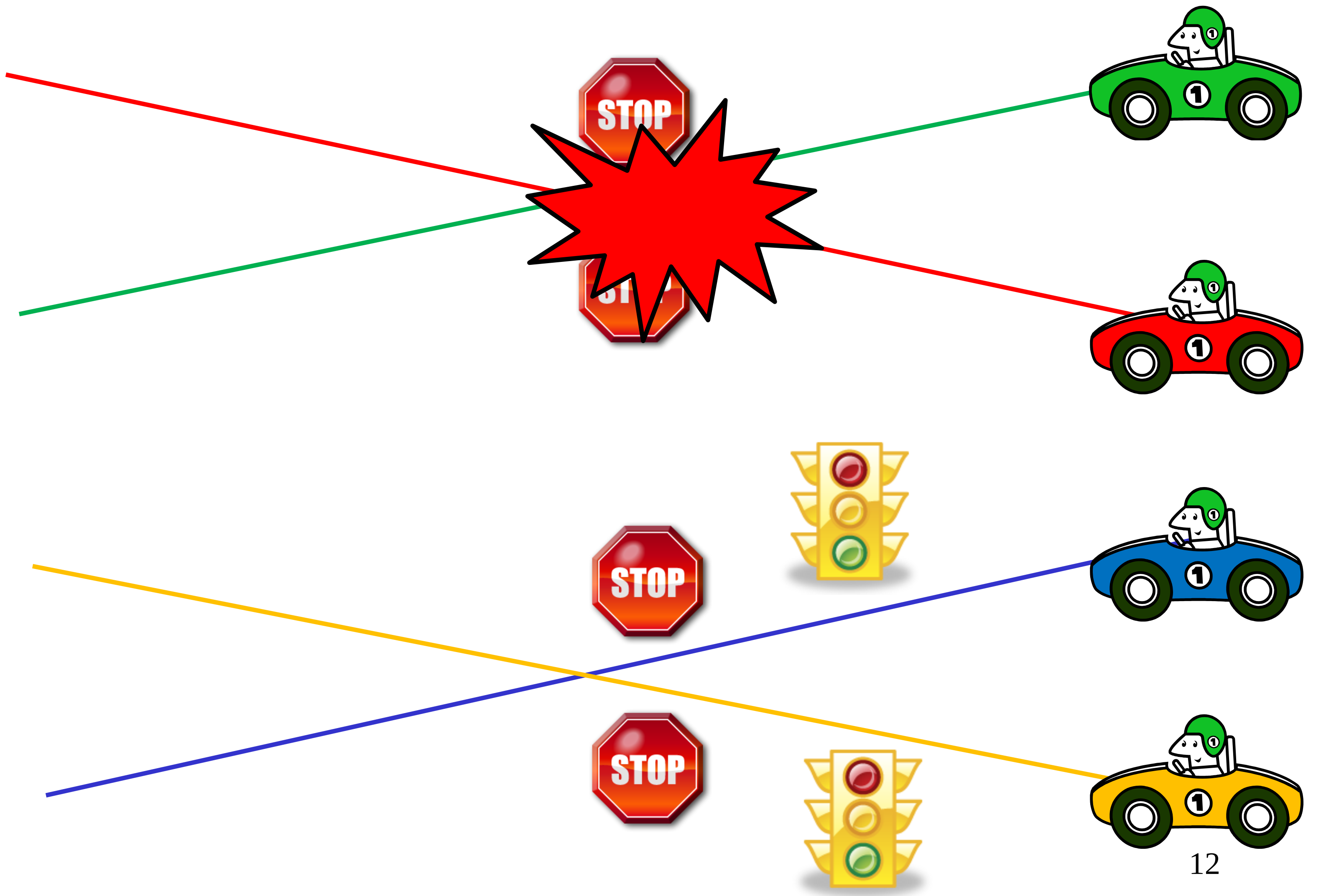
Our Approach

- Active Testing
- Phase 1: Use imprecise static or dynamic program analysis to find "abstract" states where a potential concurrency bug can happen
- Phase 2: "Direct" testing (or model checking) based on the "abstract" states obtained from phase 1

Active Testing Cartoon: Phase I



Active Testing Cartoon: Phase II



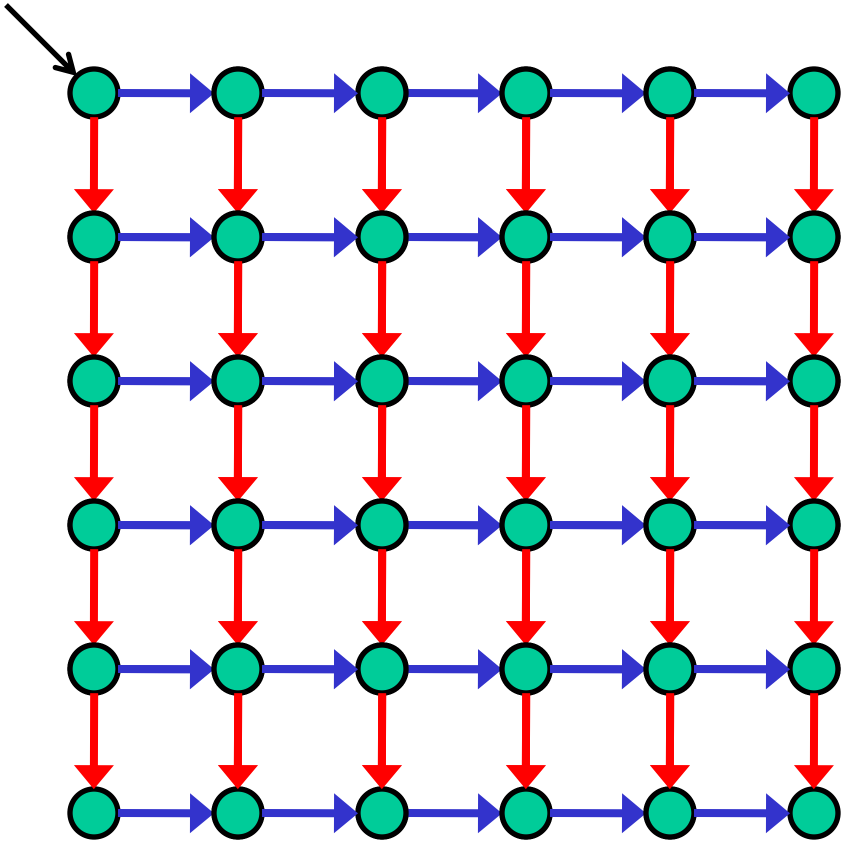
Abstract Buggy States

- A predicate on the program state
- Race: $\exists$ threads t_1, t_2 s.t. t_1 and t_2 are about to execute statements s_1 and s_2, respectively, and access the same memory location and one of the accesses is a write
- Deadlock: $\exists$ t_1, t_2 s.t. t_1 holds lock L_1 and about to acquire lock L_2 at statement s_1 and t_2 holds lock L_2 and about to acquire lock L_1 at statement s_2
- Atomicity: $\exists$ t_1, t_2 s.t. t_1 is inside an atomic block at s_1 and t_2 is about to access the same memory location at s_2
- Extensible: Define your abstract buggy state and implement custom active tester

Abstract Buggy State and Active Testing

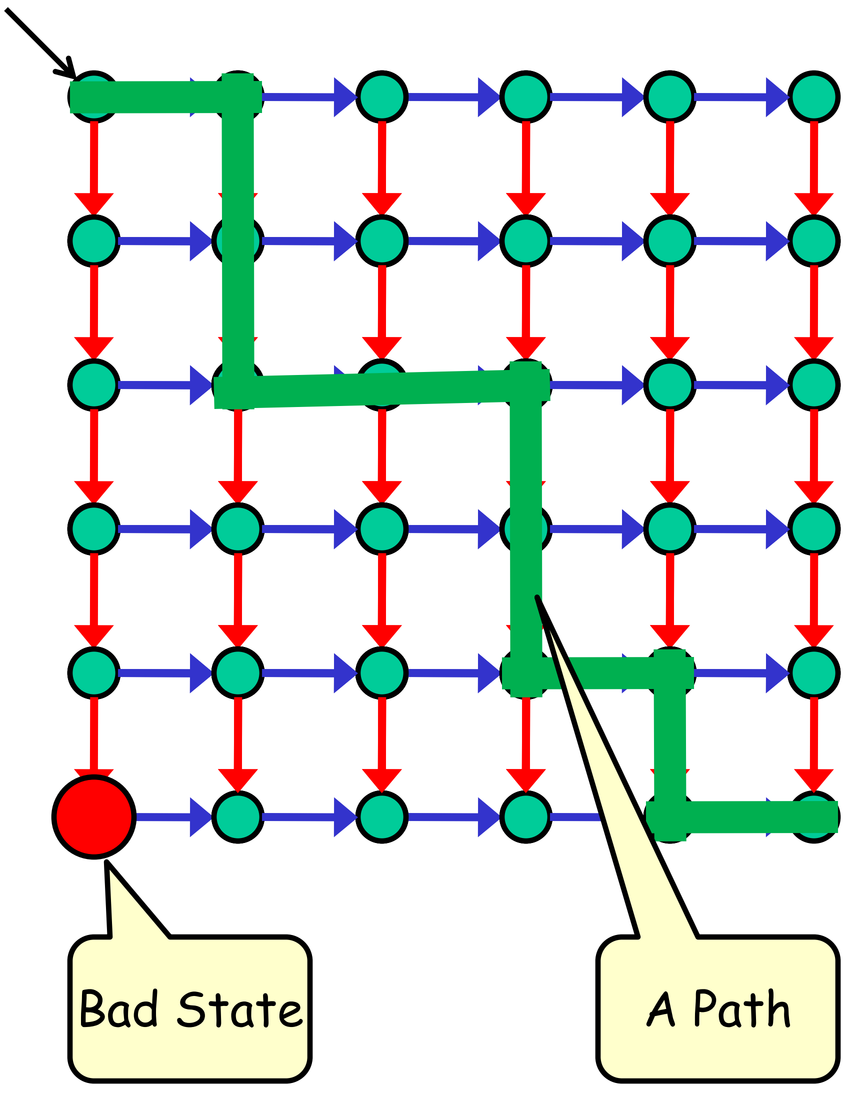
- A predicate on the program state
  - User defined
- Active Testing: Use your favorite model checker
  - But whenever a thread satisfies the abstract state predicate "partly"
    - Non-deterministically decide either to pause the thread or continue
  - We use a randomized model checker
    - But one can use Java Pathfinder or CHESS
- Summary: Add extra intelligence to your favorite model checker so that bugs get created quickly

Why it works? Simplified explanation



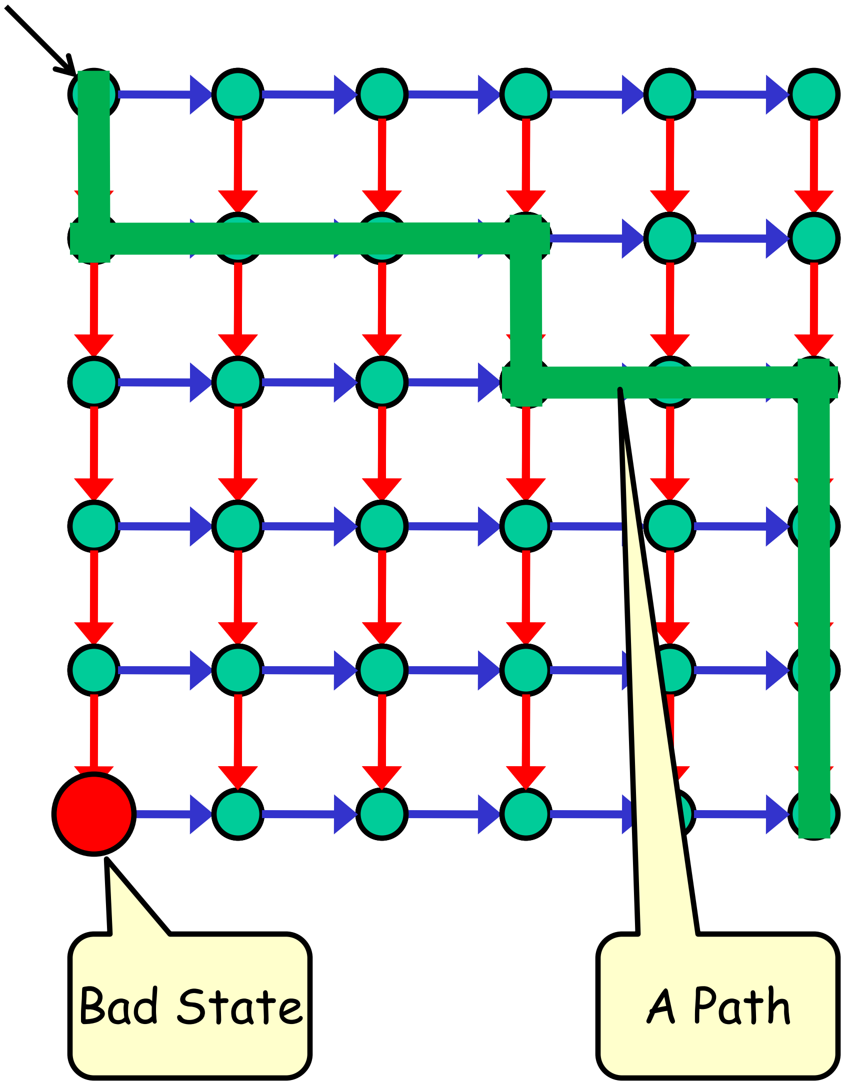
- Consider 2 threads each with n instructions

Why it works? Simplified explanation



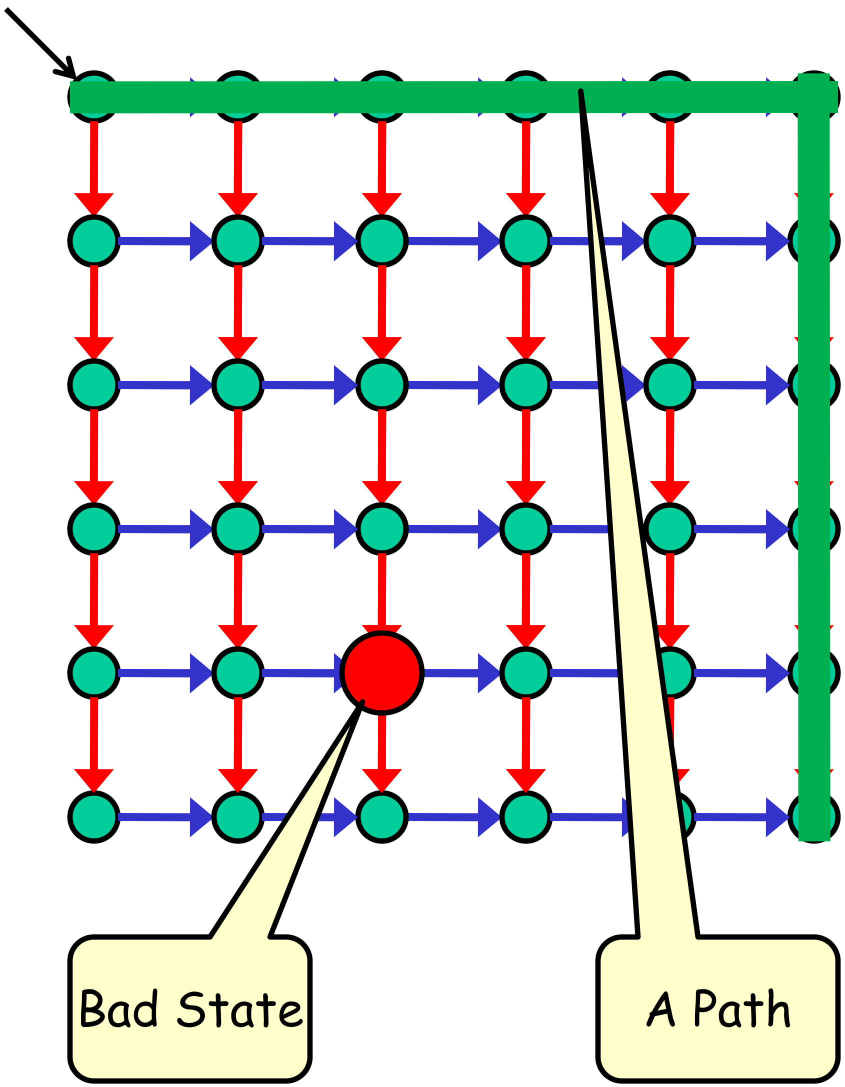
- Consider 2 threads each with n instructions
- Traditional model checker explores (2n)!/(n!n!) paths
  - Worst case probability of reaching bad state is (n!n!)/(2n)!: exponentially low

Why it works? Simplified explanation



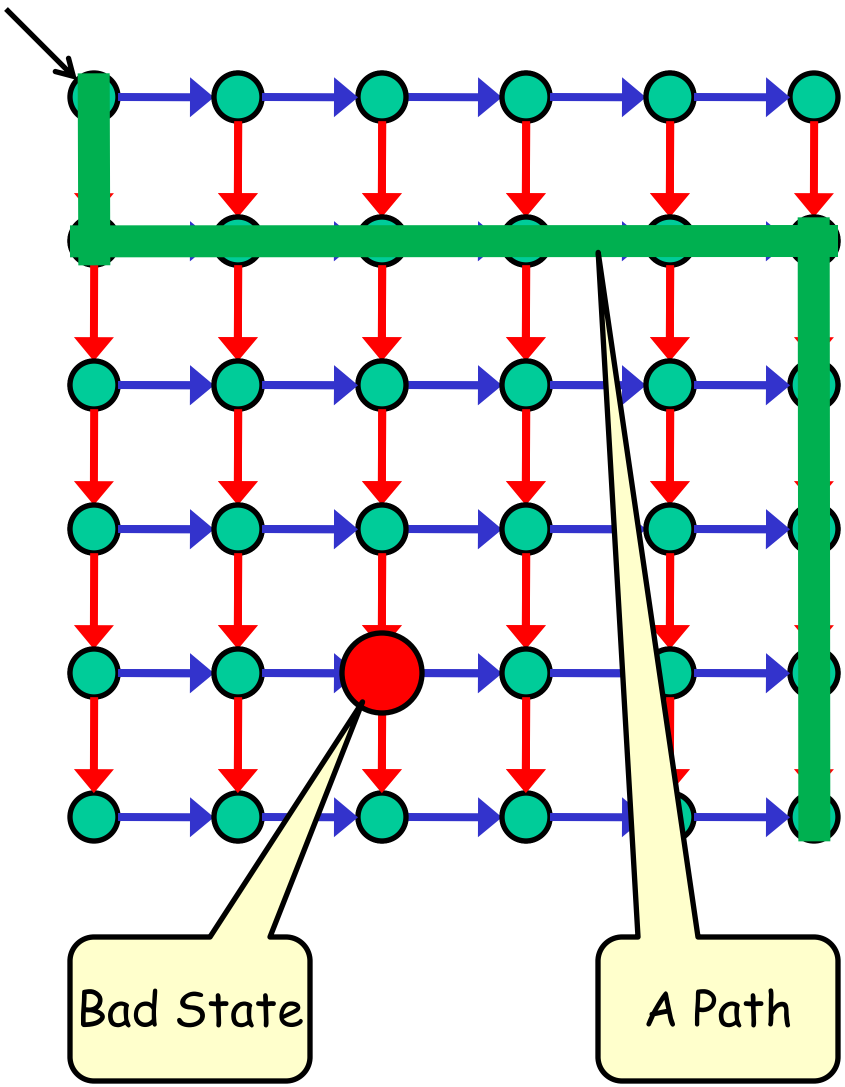
- Consider 2 threads each with n instructions
- Traditional model checker explores (2n)!/(n!n!) paths
  - Worst case probability of reaching bad state is (n!n!)/(2n)!: exponentially low

Why it works? Simplified explanation



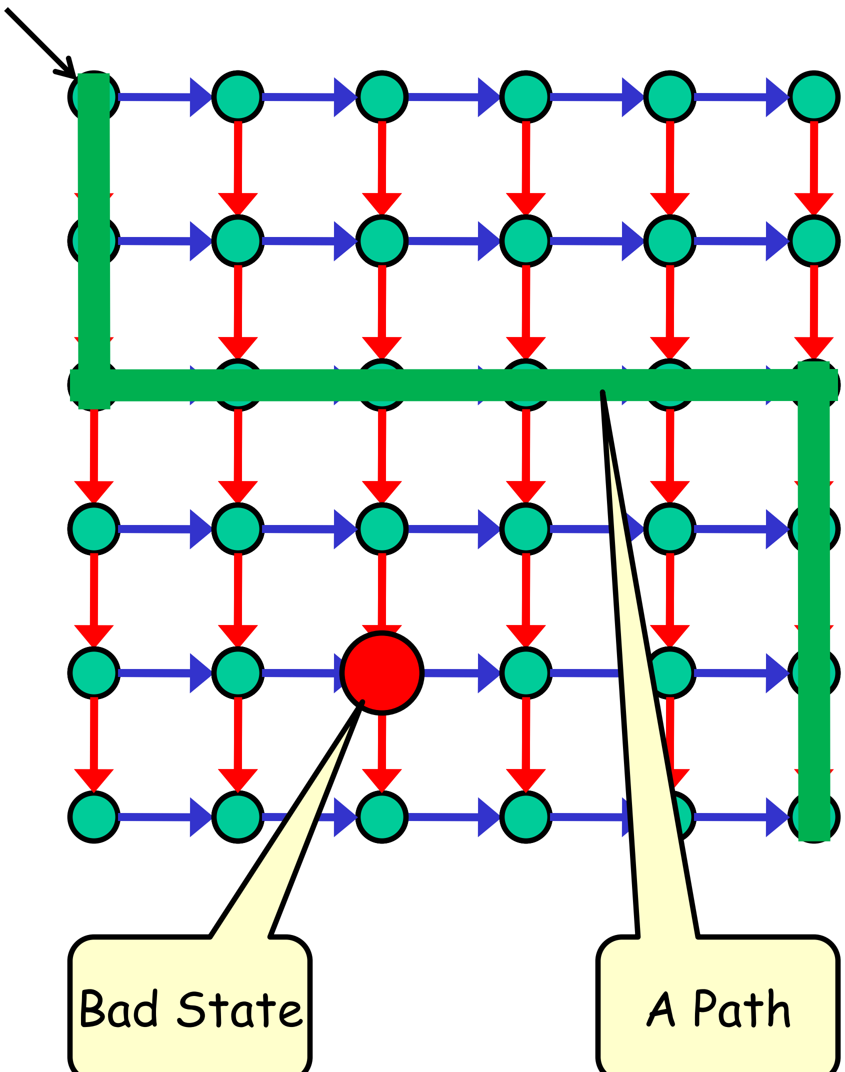
- Consider 2 threads each with n instructions
- 1-context switch bounded model checking explores 2n paths
  - Worst case probability of reaching bad state is 1/(2n): still low

Why it works? Simplified explanation



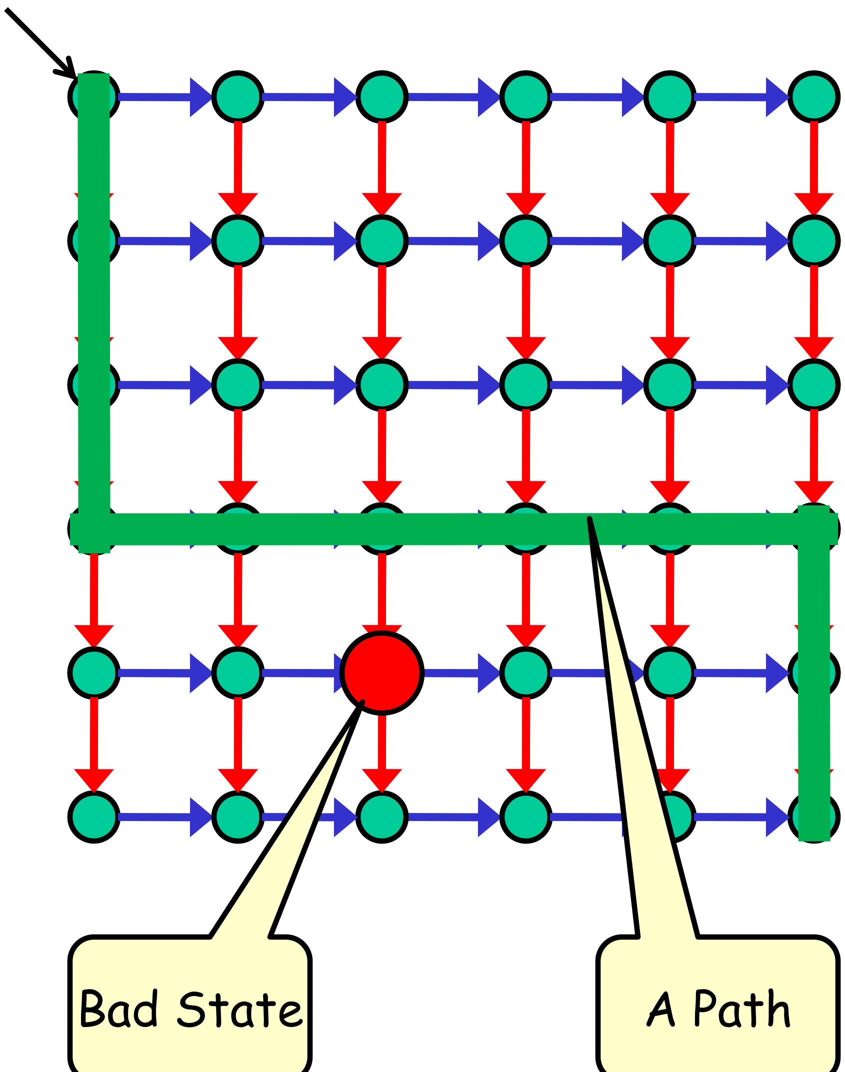
- Consider 2 threads each with n instructions
- 1-context switch bounded model checking explores 2n paths
  - Worst case probability of reaching bad state is 1/(2n): still low

Why it works? Simplified explanation



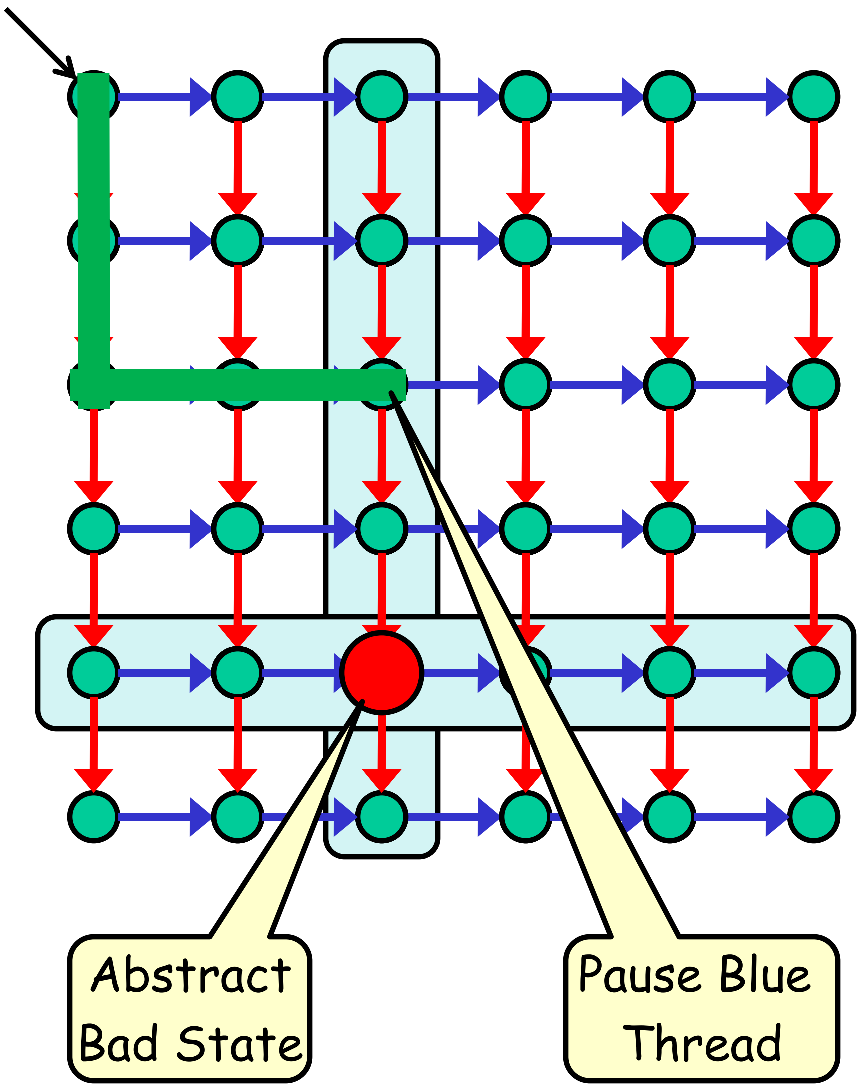
- Consider 2 threads each with n instructions
- 1-context switch bounded model checking explores 2n paths
  - Worst case probability of reaching bad state is 1/(2n): still low

Why it works? Simplified explanation



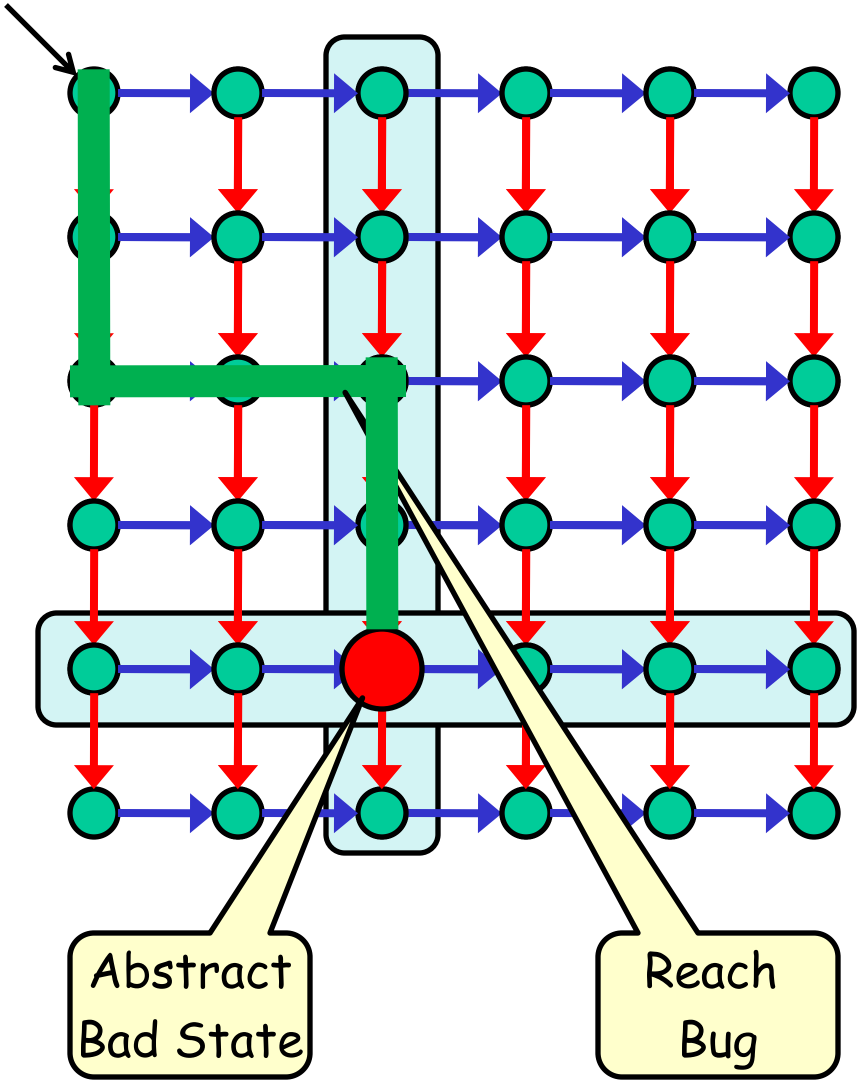
- Consider 2 threads each with n instructions
- 1-context switch bounded model checking explores 2n paths
  - Worst case probability of reaching bad state is 1/(2n): still low

Why it works? Simplified explanation



- Consider 2 threads each with n instructions
- 1-context switch bounded model checking explores 2n paths
  - Worst case probability of reaching bad state is 1/(2n): still low
- Active testing with abstraction of potential bug explores 1 schedule
  - Directed by the bug

Why it works? Simplified explanation



- Consider 2 threads each with n instructions
- 1-context switch bounded model checking explores 2n paths
  - Worst case probability of reaching bad state is 1/(2n): still low
- Active testing with abstraction of potential bug explores 1 schedule
  - Directed by the bug

Extensible Tool

- CALFUZZER for Java Programs
  - Effective random testing [ASE 07]
  - Race Directed Active Testing [PLDI 08]
  - Atomicity Violation Directed Active Testing [FSE 08]
  - Deadlock Directed Active Testing [PLDI 09]
  - User-specified pre-emption points [CAV 09]
  - Application to checking determinism [FSE 09]
- Applied to real-world programs
- Easy to implement dynamic analyses
  - Eraser, Atomizer, vector clock library, lockset, etc.
- Coming soon: THRILLE for C/C++

Summary of Bugs Found

- Races, deadlocks, atomicity violations in
  - Java Collections Framework
- Data Races found in
  - Jigsaw web server
  - weblech, hedc, Java Grande Forum Benchmark Suite (HPC)
- Deadlocks found and reproduced in
  - Jigsaw web server
  - Java Swing GUI framework
  - Java Database Connectivity (JDBC)
- Atomicity violations in
  - Apache Commons Collections

CalFuzzer in Action

C:\Windows\system32\cmd.exe

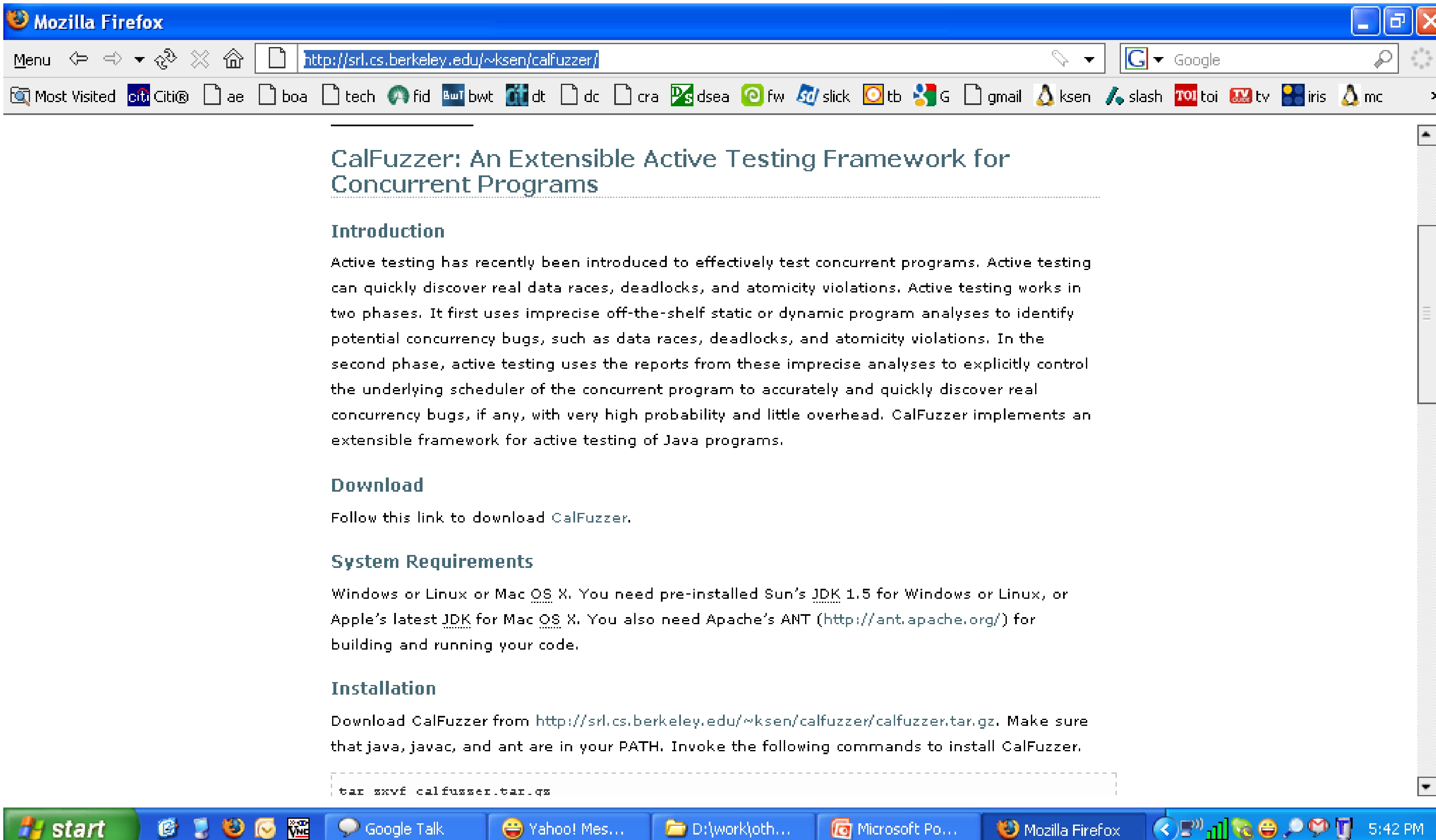
```
analysis-once:
     [java] Analysis class javato.activetesting.IGoodlockAnalysis
     [java] # of deadlocks detected 4
[stopwatch] [timer: 0.469 sec]

active-loop:
     [echo] Iteration: 1
     [echo] ------------------
     [echo] Sub-iteration: 1:1
     [echo] ------------------

analysis-once:
     [java] Analysis class javato.activetesting.DeadlockFuzzerAnalysis
     [java] cycle [[5, 7], [5, 7]]
     [java] ##############################################################
     [java] Real Deadlock Detected
     [java] ##############################################################
     [java] ##############################################################
     [java] Thread and lock sets:Thread[Thread-6,5,main]
     [java] Thread:  (7)
     [java] Thread:  (13)
     [java]     Lock  (6) at benchmarks/testcases/TestDeadlock8.java#46
     [java]     Lock  (8) at benchmarks/testcases/TestDeadlock8.java#47
     [java] Thread:  (14)
     [java]     Lock  (8) at benchmarks/testcases/TestDeadlock8.java#46
     [java]     Lock  (6) at benchmarks/testcases/TestDeadlock8.java#47
     [java] Thread:  (19)
     [java]     Lock  (6) at benchmarks/testcases/TestDeadlock8.java#46
     [java] Java Result: 1
[stopwatch] [timer: 0.360 sec]
```

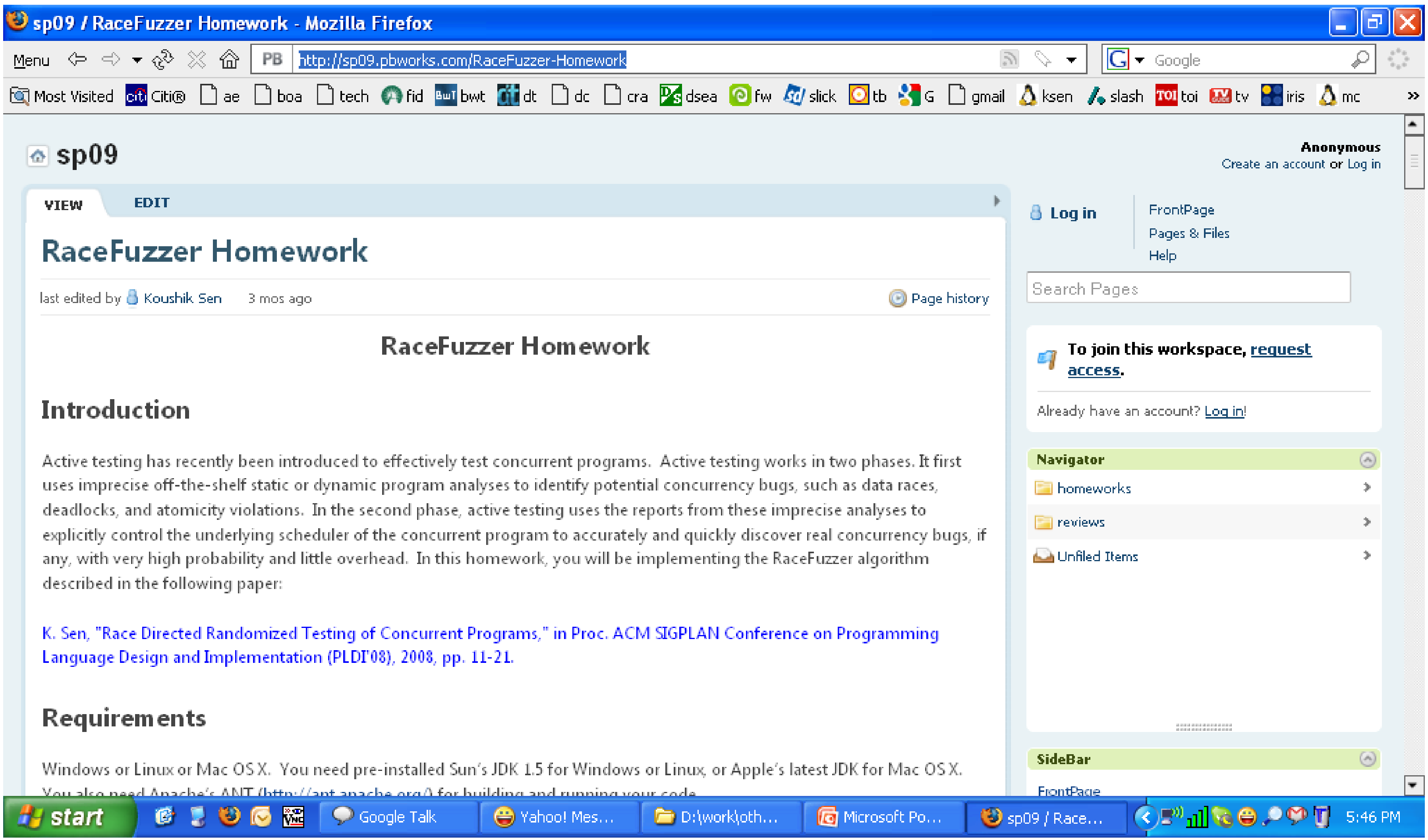
Tool for Java available for download [CAV 09]

- http://srl.cs.berkeley.edu/~ksen/calfuzzer/



Teaching Module based on CALFUZZER

- http://sp09.pbworks.com/RaceFuzzer-Homework



Conclusion

- Parallel computing will become wide-spread
  - Need testing and debugging tools
  - Because testing is what real developers use to find bugs and improve quality
- Trick is to make testing "directed" using imprecise program analyses
  - And not to make it exhaustive
- Active Testing makes concurrency testing directed
  - Confirms real bugs
  - Reproducibility is easy
  - Efficient
  - Scales really well
  - Effective