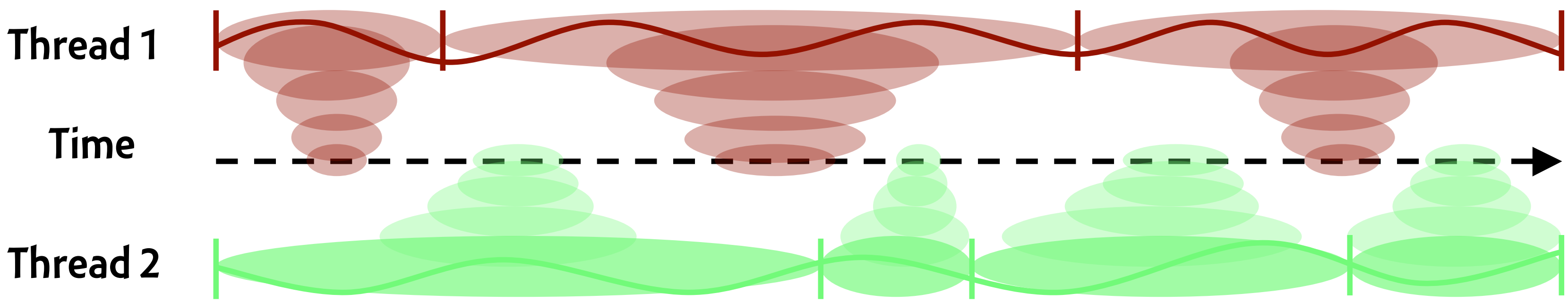
A Serializability Violation Detector for Shared-Memory Server Programs

Min Xu† Rastislav Bodík‡ Mark Hill†

†University of Wisconsin—Madison

‡University of California, Berkeley

Serializability Violation Detector: what & why$_1$



Our goals:

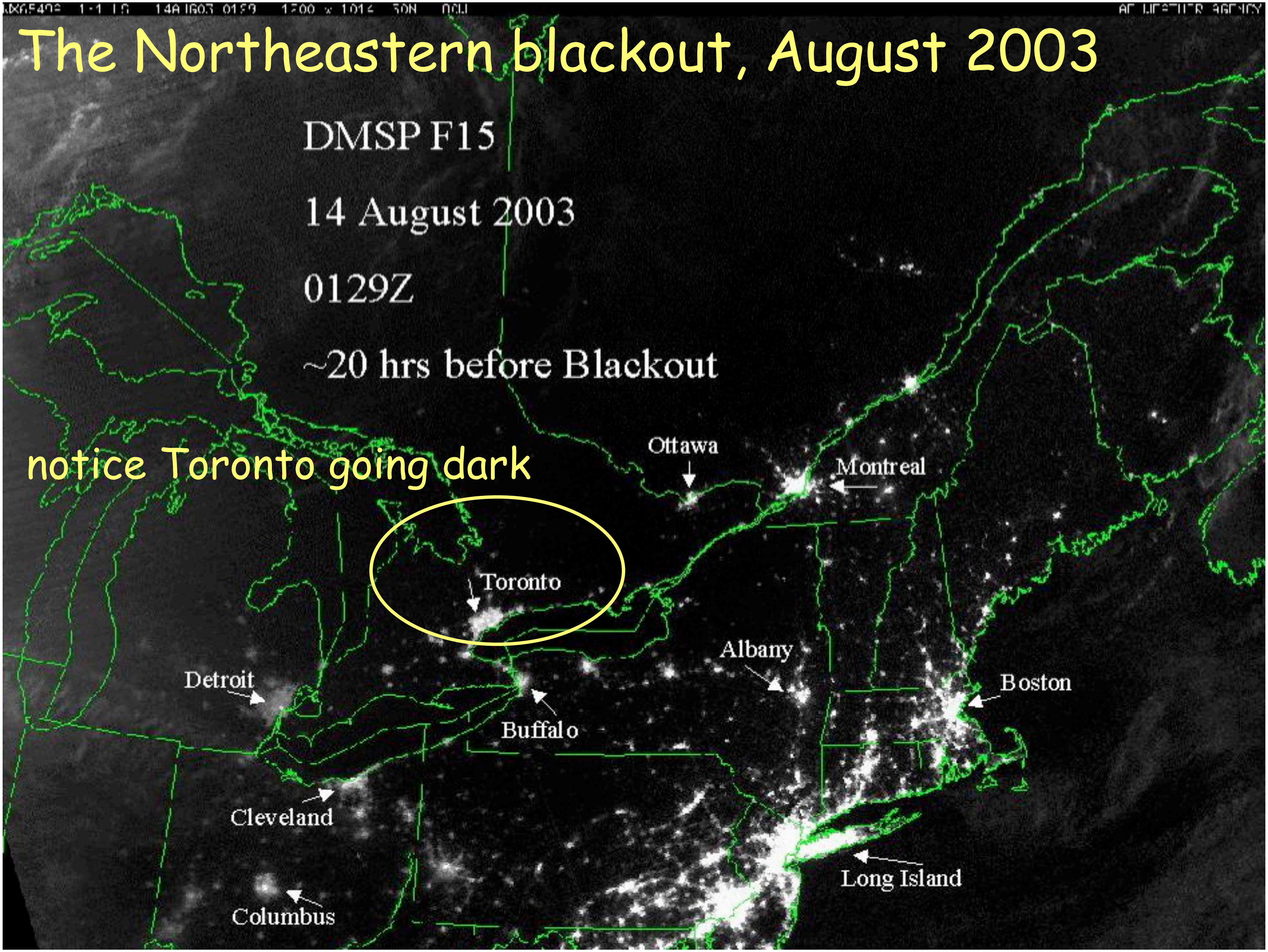
- infer code intended atomic, ⇒ avoid costly annotations
- detect bugs harmful in this run, ⇒ is this run error-free?

What's new:

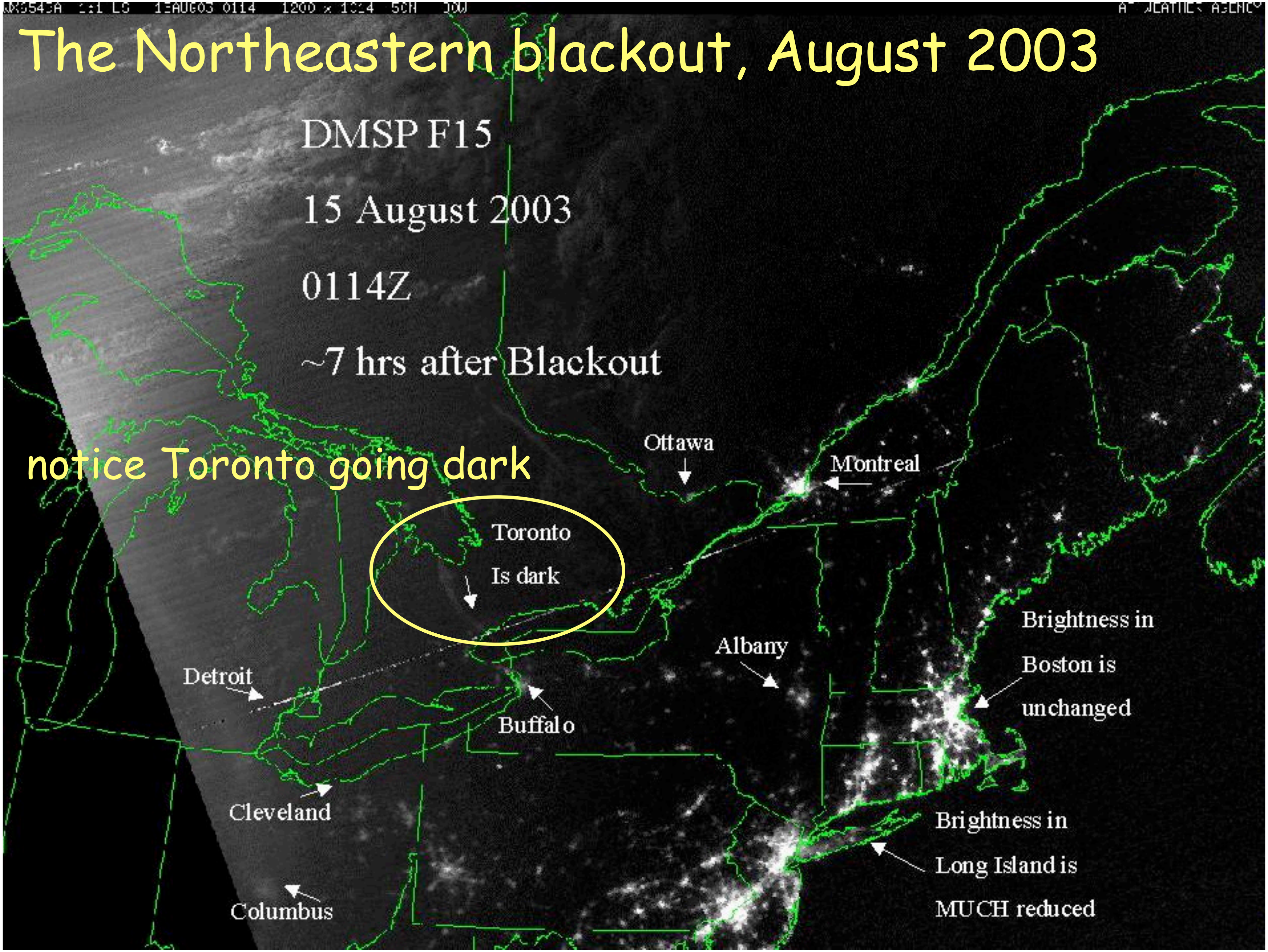
- new "atomic region", its inference and serializability

Motivation

The Northeastern blackout, August 2003



The Northeastern blackout, August 2003



Investigation revealed

… **a race condition**, triggered on August 14th by a perfect storm of events … . The bug had a window of opportunity measured in milliseconds.

SecurityFocus, April 7th, 2004

Idea:

If we could detect such harmful bugs as they occur, we could take a corrective action (e.g., reboot).

☞Recall we're interested in harmful bugs:

- bugs that actually caused an error in the observed run …
- rather than in other runs, extrapolated for coverage

Three applications of the new detector

This paper: software-based detector (slow)

1. Program debugging (post-mortem)
   - replay a failed execution, pinpoint responsible bug(s)
   - using our low-overhead deterministic recorder [ISCA'03]

Future work: hw-based detector will enable on-the-fly apps

2. Error reporting
   - Detect manifested harmful bugs, report them to operator
3. Error avoidance
   - Detect manifested harmful bugs, rollback, re-execute

How to build a detector of harmful bugs?

Q1: which correctness condition?

Q2: should we require program annotations?

Q1: which correctness condition?

1. **Datarace freedom [Netzer 93]**
   - "all conflict accesses are ordered by synchronization"
   - Neither sound, nor complete
2. **Atomicity [Flanagan-Qadeer 03]**
   - Regions are atomic if serializable in all executions
3. **Serializability**
   - *An execution is equivalent to a serial execution of all regions*

Serializability: not extrapolated to un-observed executions

Q2: require program annotations?

Annotate synchronization

- Required by datarace, atomicity detectors
- Harder without source code

Annotate atomic regions

- Required by atomicity, serializability detectors
- A burden for large legacy codes

We are *not* going to require *a priori* program annotations

Our detector

Part 1: infers code regions intended to be atomic

Part 2: checks if they are serializable

Part 1: Infer approximately the atomic regions

What guides the inference?

- Not the (potentially buggy) program synchronization
- but how shared variables are used in a computation

What is inferred?

- Our "atomic regions" are not syntactic code blocks (eg. methods)
- but the less constrained subgraphs of dynamic PDG

Correctness of inference

- Impossible to infer correctly without programmer knowledge
- our inference is a heuristic …
- … conservative if with post-mortem programmer examination

Atomic Region Hypothesis

- our hypothesis is based on empirical observations
- holds for 14 out of 14 real atomic regions examined

1. **Read shared variables, compute, write shared variables**

   i.e. all statements weekly connected via true & control dependences

2. **Shared variables are not read (again) after they are written**

   i.e. no share true dependences

Computational Units (CUs): our form of atomic regions

- subgraphs of dynamic PDG (dPDG), formed by partitioning the dPDG
- each partition is a maximal subgraph not violating the hypothesis

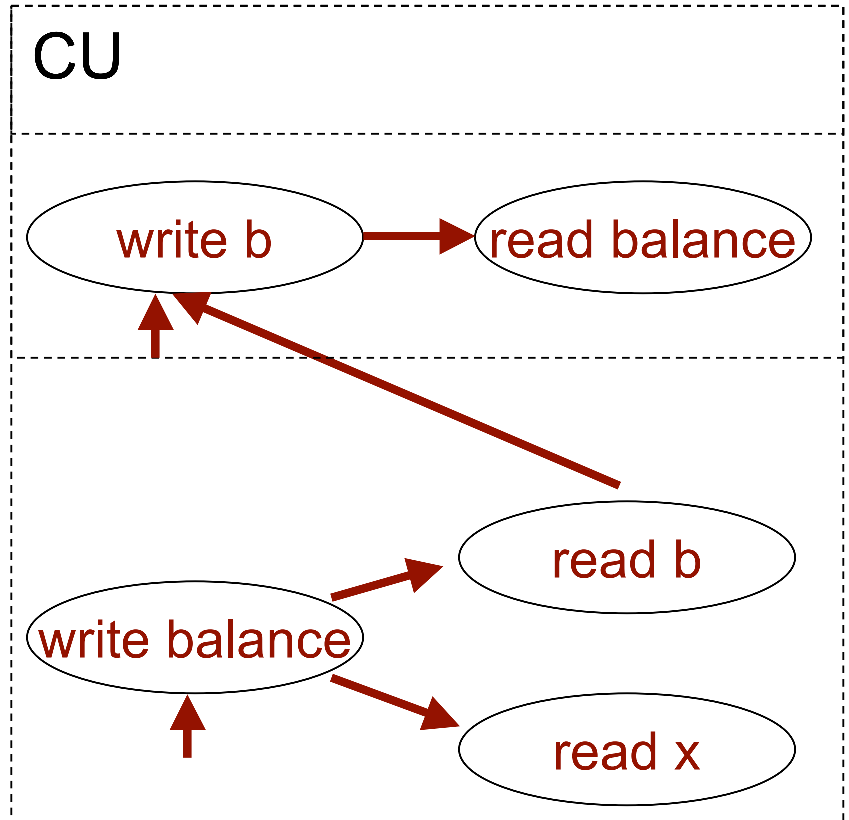
→a CU is a maximal connected subgraph not read-after-write a shared var

Example of inference


```
/* #atomic */
void withdraw(int x) {
  synchronized (this) {
    b = balance;
  }

. . .

  synchronized (this) {
    balance = b - x;
  }
}
```

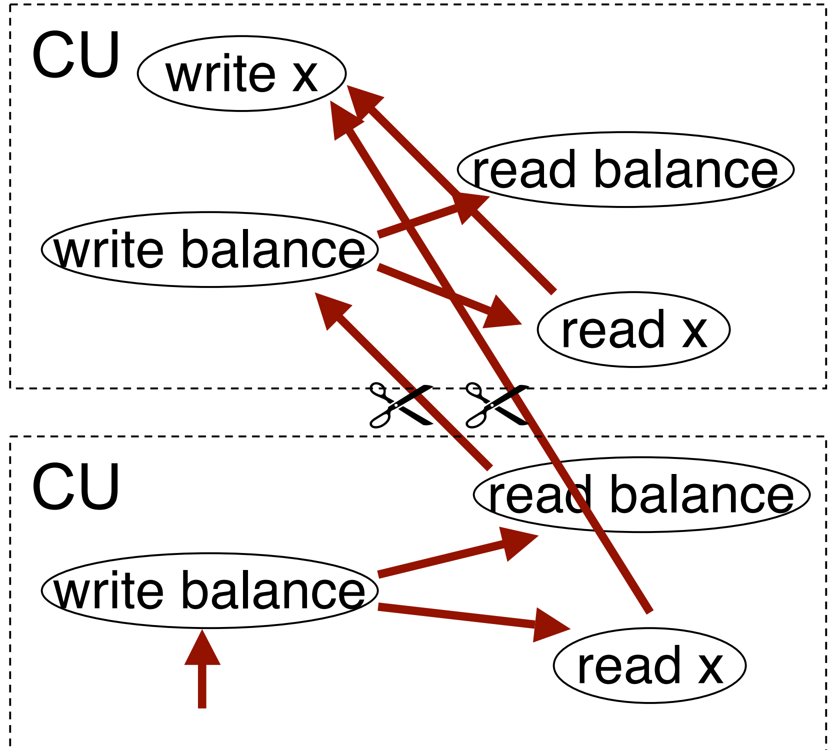


Note: CU was inferred without <u>examining synchronization</u>

Weak connection: empirically works better than backward slice

Cut dPDG into CU's


```
x = user_input();
/* #atomic */
synchronized (this) {
    balance = balance - x;
}
. . .
/* #atomic */
synchronized (this) {
    balance = balance - x;
}
```



- Partition dPDG at . . .
  . . . the starting vertices of shared true dependence arcs
- Most of the time CU's are larger than AR's, but . . .

When does inference fail?

Inferred CU <u>larger</u> than intended atomic region

- yielding false positives (spurious warnings)
- empirically, not too many

Inferred CU <u>smaller</u> than intended atomic region

- yielding false negatives (miss bugs)
- when local variables mistakenly shared (see paper)

adding soundness:

- optional post-mortem examination
- programmer examines if inferred CUs too small

Part 2: Detect serializability violations

Check for stale values at the end of a CU

CU

```
void withdraw(int x) {
  synchronized (this) {
    b = balance;
  }


  synchronized (this) {
    balance = b - x;
  }
}
```

```
void deposit(int y) {
  synchronized (this) {
    balance = balance + y;
  }
}
```

More details in the paper

Experimental evaluation

Experiment setup

Full-system simulator

- GEMS: http://www.cs.wisc.edu/gems
- 4 processors; 4-wide OoO; 1GHz; 4GB; SPARC; Solaris 9
- Detector transparent to OS & applications
- Simulator provides deterministic re-executions

Benchmarks (Production environment setups)

- Apache (w/o, w one known harmful bug)
- MySQL (w/o, w one unknown harmful bug)
- PostgreSQL (w/o any known bug)

Frontier Race Detector (FRD)

Goal: compare SVD with a datarace detector

- We developed it to avoid *a priori* annotations

FRD

- Multi-pass detection
  - Run1: frontier races = synchronization races ∪ dataraces
  - Manually separate synchronization races and dataraces
  - Run2: find remaining dataraces

Performance metrics

False negatives

- Silent on harmful bugs

Dynamic false positives

- Spurious reports -- including duplicates
- bad for online reporting or avoidance applications

Static false positives

- Spurious reports -- not including duplicates
- bad for offline debugging applications

Overhead

- Time & space

False negatives

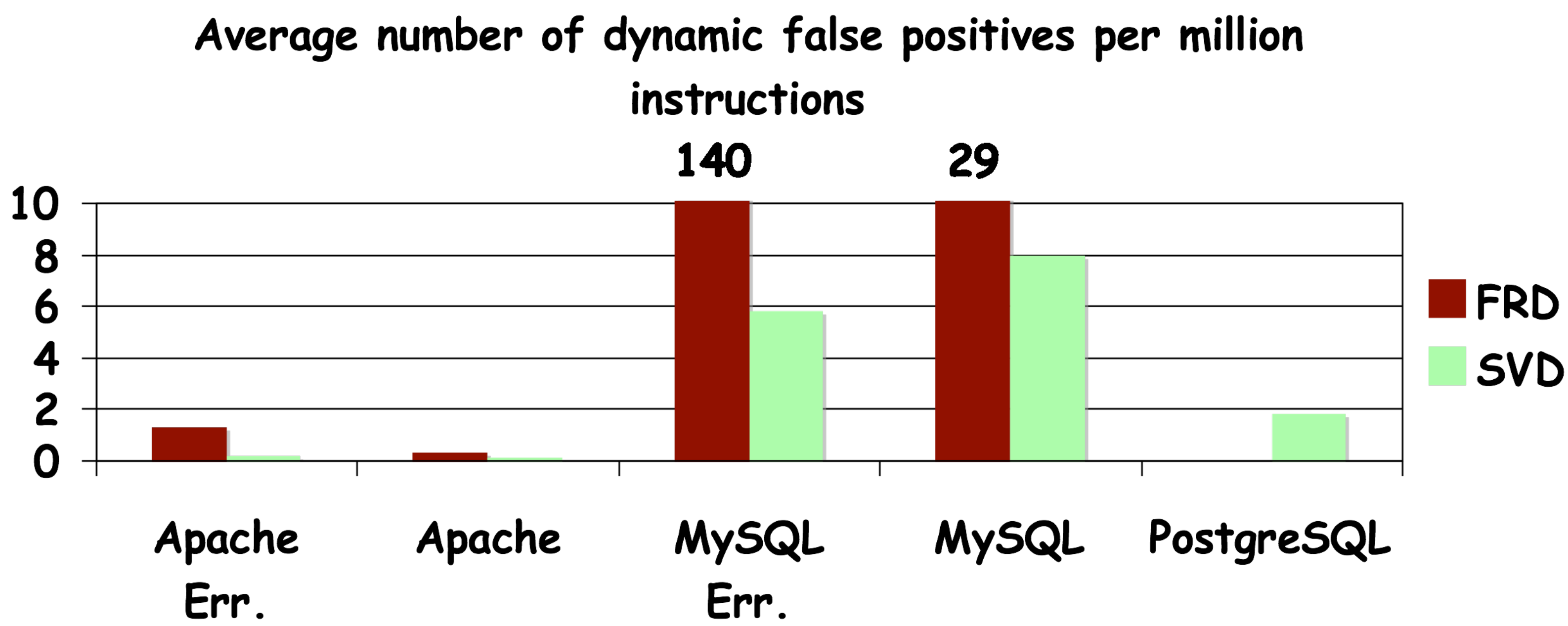
No false negatives:

- SVD: detect Apache buffer corruption (on-the-fly)
- SVD: find root cause of a MySQL crash (post-mortem)
- FRD: find the bugs since both are dataraces (post-mortem)

SVD: Soundness provided by post-mortem examination

- No *a priori* annotation effort required
- The post-mortem examinations take
  - ~0.5 hour for Apache
  - ~10 hours for MySQL
  - ~1 hour for PostgreSQL

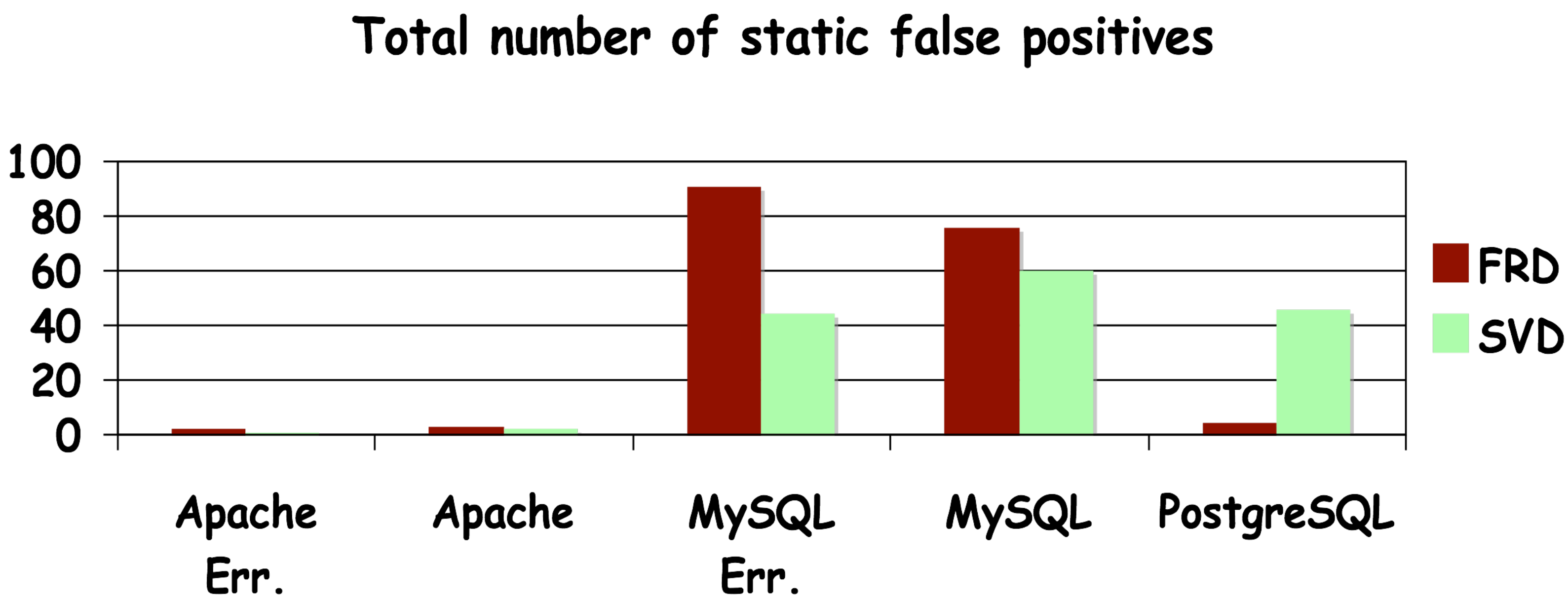
Dynamic false positives



SVD is better than FRD in finding harmful bugs

- datarace detectors reports errors in correct executions
- far fewer dynamic false positives for Apache and MySQL

Static false positives



- PostgreSQL: mature software, testing removed dataraces?

SVD overhead

Overhead is high

- Time: up to 60x
- Space: up to 2x
- Room for improvement exists
  - Compiler computes PDG
  - Seeking hardware implementation

Eventually, should have a low overhead hardware detector

Conclusion

A dynamic detector for harmful bugs

- 1. Program debugging
- 2. Error reporting
- 3. Error avoidance

SVD

- No *a priori* program annotations
- Finding harmful bugs in failing runs

Thank you!