Debugging Concurrent Java Programs by Minimizing Scheduling Noise

Yaniv Eytani and Timo Latvala

University of Illinois at Urbana-Champaign

Haifa Verification Conference 2006

Outline

- Concurrency and noise making
- Known bug patterns and observations
- Minimizing scheduling noise
- Lessons learned and future work

Concurrent methodologies and tools?

- Multi-core architectures becomes widespread
  - Concurrent programming is necessary and common
- Non-determinism is hard to analyze and debug
  - Race conditions and deadlocks are common, yet difficult to uncover

Noise making?

- For a given functional test, the set of interleavings is possibly unbounded
  - Only a fraction of the interleavings actually produce race conditions and deadlocks
- Generate biased interleavings that produce race conditions
  - Introducing conditional context switches and timeouts during the program's execution

A simple example

- Initially x=0

thread1	thread2	thread3
x++;	x++;	x++;

- Three threads are incrementing "x" by 1; what are the possible outcomes?

x=1 or x=2 or x=3

A simple example (Continued)

global x=0

.

.

global x=1

.

.

global x=2

.

.

global x=3

Thread1	Thread2	Thread3
read x=0		
local x++		
write x=1		
	read x=1	
	local x++	
	write x=2	
		read x=2
		local x++
		write x=3

A Simple example (Continued)

global x=0

.

(global x=0)

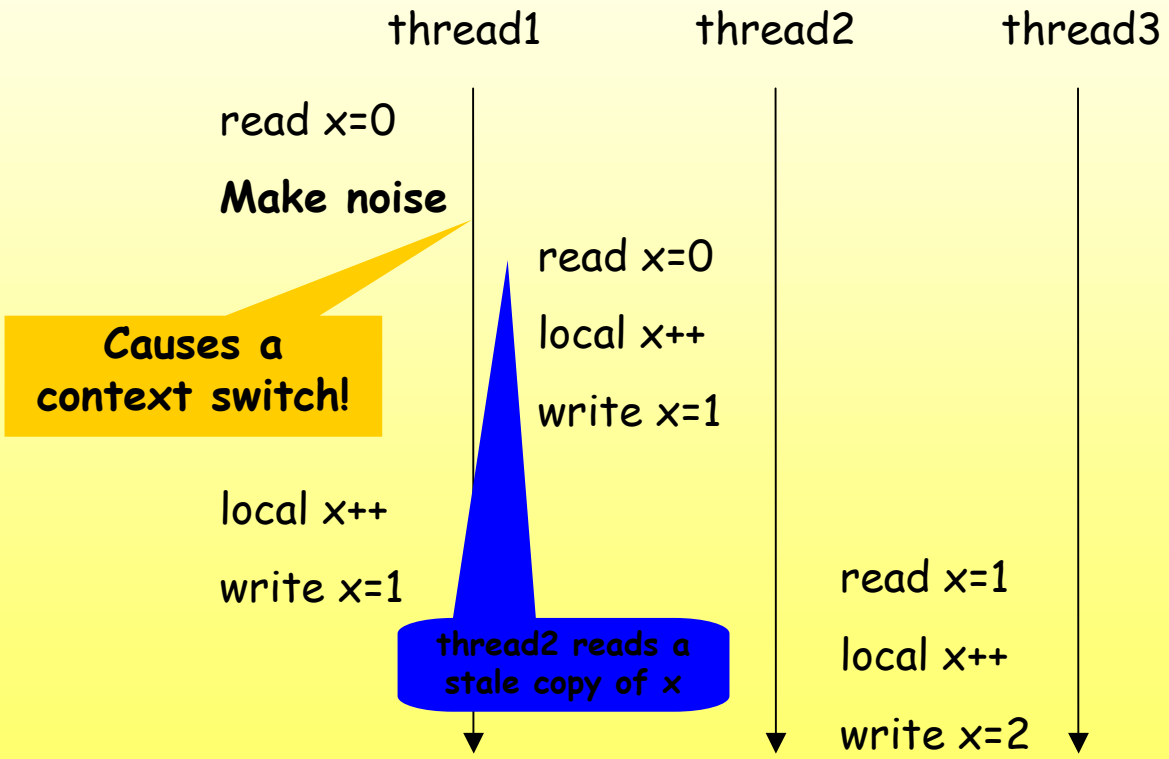
global x=1

global x=1

.

.

global x=2



Known bugs patterns are usually small

- Concurrent bugs surveyed are usually:
  - Forms of a non-atomic execution
    - Data races and atomicity violations
  - Deadlocks
    - On a shared resource or communication (wait/notify etc.)
- Most involve only a specific ordering of a few events

Some observations

- "Noising" at different locations
  - Increases or decreases the probability a bug manifests
- Only small amount of focused noise required
  - For the bug to appear frequently
- Noised locations are indicative
  - To source code locations "related" to the bug

Where is the bug ? - I

T1	T2
A1	A2
B1	B2
C1	C2
D1	D2
E1	E2
F1	F2

T1	T2
A1	A2
B1	B2
C1	C2
D1	D2
E1	E2
F1	F2

- $X_i \longrightarrow X_j$
- Means a context switch occurred in the trace

Two traces that lead to the bug manifesting

Where is the bug ? - I

T1	T2
A1	A2
B1	B2
C1	C2
D1	D2
E1	E2
F1	F2

T1	T2
A1	A2
B1	B2
C1	C2
D1	D2
E1	E2
F1	F2

- $X_i \longrightarrow X_j$
- Means a context switch occurred in the trace

Two traces that lead to the bug manifesting

Where is the bug? - II

T1	T2
A1	A2
B1	B2
C1	C2
D1	D2

1) E1: If (x !=0) → 2) E2: x=0

3) F1: y=1/x F2

1. Simple precondition
   - Reach 1 hb -> 2
2. One necessary switch
3. Easily "Explains" the bug !

Interactions within the program

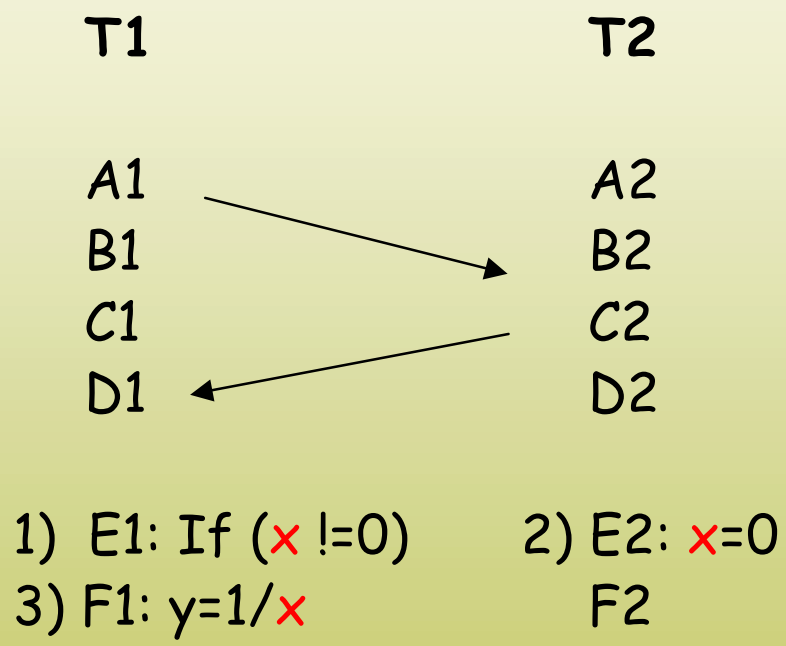
T1	T2
A1	A2
B1	B2
C1	C2
D1	D2

1) E1:lock(a) → 2) E2: lock(b)

3) F1: lock(b) ← 4) F2: lock(a)

1. "Choose" only:
   - 1 hb -> 4
   - 2 hb -> 3
- Depends on the precondition
- Choosing at random
  - High probability

Removing unrelated switches



- If we start at T1:
  - Would like to reach
    - 1 -> 2
- Need even number of switches before E1!
- Cannot arbitrarily remove "unrelated" switches
  - To the non - atomic block

Minimizing Scheduling noise

- We would like to debug basing on:
  - Given a functional test + set of noises
    - Exhibits a concurrent bug
  - Find to find a small set of locations
    - Where "noising" causes the bug to manifest frequently

Motivating model

- A simple model for choosing relevant program locations:
  - **Good locations** – noise increases the probability that the bug manifests
  - **Bad events** –noise decreases the probability the bug manifests
  - **Neutral events** –noise does not greatly impact the probability the bug manifests

A simple approach

- Problem, lack of knowledge:
  - "Good" set of locations is unknown
  - Entities within the program interact
    - Locations may alternate between "neutral" and "bad"
- Solution:
  - Use a randomized approach
    - Works well if:
      - interactions are small
      - Small set of locations

Simple Black box algorithm

1. Use a noise maker to obtain a failed run
   - Record the "noised" program locations

1. Choose a subset of these locations
   - Repeat step 1, limit the set of possible locations
     - To the set of chosen location at step 2

- Stop when:

- Small set of locations obtained
- Bug manifests frequently

Tomcat Apachie Logger

```
public static void startCapture() {
...
    if (!reuse.empty()) {
        log=(CaptureLog) reuse.pop();
    }
    else {
        log=new CaptureLog();
    }
... }
```

A race may cause an **EmptyStack-exception**

Crawler I

Thread 1:

```
if(connection ! = null)
   connection.setStopFlag( );
```

Thread 2:

```
connection  = null
```

Throws a null pointer exception

Crawler II

- When a thread is seeded with a delay near the end of the run
- finish() executes before the waitForConnection().
  - Leads to the program not terminating
- Examining printings used for debugging
  - Easily allows to explain the bug

Additional code used

- Other programs experimented with:
  - Java runtime libraries
  - Readers/Writers Implementation
  - Set of benchmark programs
    - Mostly of small size
    - Contain documented known bug patterns

Explaining the bug

- Code is many times written in a concise way
  - Knowing a "good" switch localize the bug
- A Simple review of the trace helps
  - Next access(es) to the same variable from other threads
    - Also very indicative for localizing
  - Relevant methods
    - Usually provide a good high level description

Current limitations and future work

- Focus only at one bug each time
  - Common methodology
- Works only with intermittent bugs
  - Use also “bad” noise in the future!
- Needs more specialize heuristics
  - To allow explaining more complex scenarios
    - E.g., noise at granularity of methods

Acknowledgments

- Eitan Farchi
- Shmuel Ur
- Yosi Ben-Asher
- Rajesh Kumar
- Feng Chen
- Koushik Sen

Q&A?

Thank You !

The Interleaving Space - Terminology

- Access to shared variables and synchronization operations are called critical events
- A thread schedule is determined by the order of critical events occurrence
  - A possible thread scheduling is called an interleaving
  - The set of all possible interleavings is called the interleaving space

“Bad” noise can mask the bug

Global x=0

.

.

Global x=0

.

.

Global x=1

.

.

Global x=2

Making noise Causes a context switch!

The noise on `y` cancel the effect of the noise on `x`

Thread1	Thread2
Read X=0	
Local X++	
Make noise	
	Write Y=1
	Make noise
Write x=1	
	Read X=1
	Local X++
	Write x=2