

## Detecting and Fixing Concurrency Bugs

Shan Lu  
University of Chicago



## About course assignment

- Study 5 bugs in an open-source application's Bugzilla
  - Pick the keyword you like
  - Pick the application you like (or use cbs ...)
  - Write the following for each bug
    - What is the bug root cause (fault)
    - What errors might be caused by the bug
    - What is the failure symptom of this bug
    - What is the fix strategy of developers
    - Can this bug be automatically detected? Exposed during testing? Automatically diagnosed or fixed?
- You can work in group

2

Types of bugs

constraints

Fighting approaches

3

## Different types of bugs

- Memory bugs
  - Memory leaks
  - Buffer overflow
  - Null-ptr dereference
  - Uninitialized read
- Semantic bugs
- **Concurrency bugs**
- Performance/energy bugs

Faults in linux: ten years later. ASPLOS'11  
Bug Characteristics in Open Source Software. EMSE'13

4

Don't hesitate to ask me  
questions!

5

## Background

Thread  
Concurrency Bugs

6

## Thread vs. Process

- Process – resource management unit
  - Nothing is shared among processes, except ...
  - Parent & child share initial image
- Thread – execution/scheduling unit
  - The address space is completely\* shared among threads under the same process

*See example code*

7

## Sources of non-determinism

- race.c
- On single-core machines
  - System event non-determinism
- On multi-core machines
  - System event non-determinism
  - (Parallel) hardware on-determinism

8

## Thread synchronization (I)

- Lock
  - Enforce mutual exclusion
- Condition variable
  - Enforce pair-wise ordering
- What is needed to synchronize ...?
  - (1) *Thread 1* X++;                      *Thread 2* X++;
  - (2) *Thread 1* p=malloc(10); *Thread 2* \*p=10;

9

## Thread synchronization (II)

- Semaphore
  - A counter (can be initialized with any positive value)
  - P (acquire one piece of resource)
  - V (release one piece of resource)
- What is needed to synchronize ...?
  - (1) *Thread 1* X++;                      *Thread 2* X++;
  - (2) *Thread 1* p=malloc(10); *Thread 2* \*p=10;

10

## Thread APIs

- pthread\_create
- pthread\_join
- pthread\_mutex\_lock
- pthread\_mutex\_unlock
- pthread\_cond\_wait
- pthread\_cond\_signal
- ...

11

## Other way of parallel execution

- Shared memory vs. message passing

12

### What are concurrency bugs?

- Untimely accesses among threads (buggy interleavings)

Thread 1      Thread 2

```

if (proc){
  tmp="proc";
}
        
```

MySQL

Thread 1 (child)      Thread 2 (parent)

```

mThd=CreateThd();
_state = mThd->state;
        
```

Mozilla

13

### It is important to fight con. bugs

14

### Different aspects of fighting bugs

15

### Outline

- What are concurrency bugs
- Concurrency bug detection
- Concurrency bug exposing
- Concurrency bug fixing
- Others
- Conclusion

16

### Outline

- What are concurrency bugs
- Concurrency bug detection
- Concurrency bug exposing
- Concurrency bug fixing
- Others
- Conclusion

17

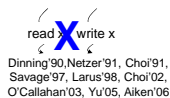
### The key challenges

- What type of interleavings is buggy?
- Large state space
- False positives
- False negatives
- Overhead

18

## Data race

- Definition



- Does this pattern match our examples?
- How to get rid of a data race?

19

## How to detect data races?

- How do I know the execution of two accesses are concurrent?

- What does basic run-time monitoring tell us?

count ++; <thread 1>

... //millions of instructions in between

count++; <therad 2>

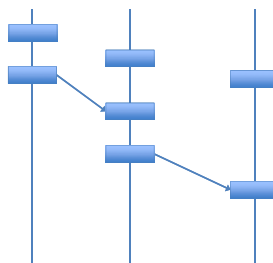
20

## Physical time vs. logical time

- From Leslie Lamport
- What ordering do we know for sure in a distributed environment?
- Logical time based on causality/happens-before relationship
  - Vector timestamp
  - Scalar timestamp

21

## What ordering is guaranteed?

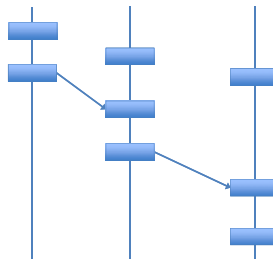


## Logical time

- Operations within one thread are (happens-before) ordered following program semantics
- Message sending is (happens-before) ordered before message receiving
- Ordering is transitive
  - $A \rightarrow B, B \rightarrow C \Rightarrow A \rightarrow C$

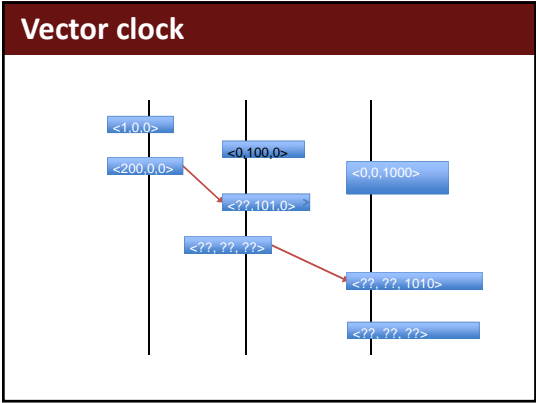
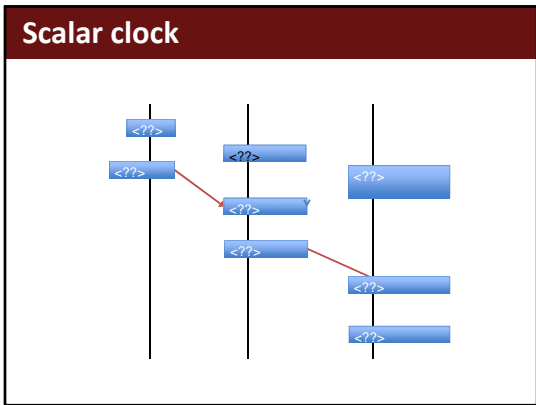
23

## How to represent logical time?



### (scalar) logical clock

- Design a clock that can reflect the happens-before order
  - Increment within one process
  - Increment when receiving a message



### How to use logical time in race det?

- What is the causality relationship here?
- Example 1
 

<i>Thread 1</i>	<i>Thread 2</i>
tmp=x;	tmp=x;
x = tmp+1;	x=tmp+1;
- Example 2
 

<i>Thread 1</i>	<i>Thread 2 (child)</i>
p=malloc(10);	*p=10;
pthread_create(...)	
- Example 3 (lock)

### How to detect data races?

- Happen-before algorithm
  - Use logic time-stamps to find concurrent accesses

Thread 1	Thread 2
	lock (L); <0,1>
	ptr=NULL; <0,2>
	unlock(L); <0,3>

```

<1,0>ptr = malloc(10);
<2,3>lock (L);
<3,3>ptr[0]='a';
<4,3>unlock(L);
    
```

### How to detect data races?

- Happen-before algorithm
  - Use logic time-stamps to find concurrent accesses

Thread 1	Thread 2
	ptr=NULL; <,>
	barrier(&b); <,>

```

<,> barrier(&b);
<,> ptr = malloc(10);
<,> ptr[0]='a';
    
```

## How to detect data races?

- Happen-before algorithm
  - Use logic time-stamps to find concurrent accesses

```

Thread 1          Thread 2
<1,0>ptr = malloc(10);
<> lock (L);
<> ptr[0]='a';
<4,0>unlock(L);

                lock (L); <4,1>
                ptr=NULL; <4,2>
                unlock(L); <,>

```

31

## Happen-before algorithm summary

- Strength
  - Work for different types of synchronization
  - Few false positives in race detection
- Weakness
  - False negatives in race detection

32

## How to detect data races?

- Lock-set algorithm
  - A common lock should protect all conflicting accesses to a shared variable

```

Thread 1          Thread 2          Thread 1          Thread 2
                lock (L);          ptr = malloc(10);          lock (L);          lock (L);
                ptr=NULL;          <L>          lock (L);          ptr=NULL;          <L>
                unlock(L);          <L>          ptr[0]='a';          unlock(L);          <L>
                <L>          ptr = malloc(10);          <L>          <L>
                lock (L);          <L>          lock (L);          ptr=NULL;          <L>
                ptr[0]='a';          <L>          ptr[0]='a';          unlock(L);          <L>
                unlock(L);          <L>          unlock(L);          <L>

```

Eraser : A dynamic data race detector for multithreaded programs, TOCS'97 33

## Lock-set algorithm summary

- Strength
  - Fewer false negatives
    - Interleaving in-sensitive
- Weakness
  - More false positives
  - Cannot handle non-lock synchronization
- How to solve the false positive problem?
  - H-B & Lockset hybrid race detection

RaceTrack: efficient detection of data race conditions via adaptive tracking, SOS'05 34

## Are we done?

- Performance
  - Huge problem
  - Solution?
- False positives
  - Huge problem
    - 90% of data races do not lead to visible failures\* [PLDI'07]
  - Solution?
- False negatives

```

Thread 1          Thread 2
ptr = malloc(10); lock (L);
lock (L);          ptr=NULL;
ptr[0]='a';        unlock(L);
unlock(L);

```

35

## How to speed-up?

- Hardware support
  - Non-existing
  - Existing
- Sampling

36

## Break

37

## How to do better?

Let's find a more accurate root-cause pattern for concurrency bugs!

38

## Root-cause patterns

- A study of 105 real-world concurrency bugs

Learning from Mistakes --- A Comprehensive Study on Real World Concurrency Bug Characteristics, ASPLOS08

## Root-cause patterns

Thread 1

```
if (proc){
  tmp="proc,"
}
```

MySQL

Thread 2

proc = NULL;

**Atomicity Violation Bugs**

70%

Thread 1 (child)

Thread 2 (parent)

mThd.CreateThd();

\_state = mThd->state;

Mozilla

**Order Violation Bugs**

30%

40

## Root-cause patterns

*Multiple Variable*

30%


*Single Variable*

70%

70%

41

## Why did I do this study?



42

### How to detect atomicity-violations?

- Problem 1
  - Know which code region should maintain atomicity
- Problem 2
  - Judge whether a code region's atomicity is violated

43

### How to detect atomicity-violations?

- Problem 1
  - Know which code region should maintain atomicity
- Problem 2
  - Judge whether a code region's atomicity is violated

```

READ flag
READ flag
while (!flag)
READ flag
...
                    
```

flag = 1;

```

Thread 1      Thread 2
if (proc){    proc = NULL;
tmp=*proc;   }
}
MySQL
                    
```

44

### Solution to problem 2

- Atomicity violation = unserializable interleaving

Thread 1

```

access x
atomic ?
access x
                    
```

Thread 2

```

-accesses
                    
```

Read x

```

atomic
Read x
                    
```

Write x

```

not atomic
Read x
Write x
                    
```

- Totally 8 cases of interleaving

Read x Read x	Write x Read x	Read x Read x	Write x Write x
Read x Write x	Write x Read x	Read x Write x	Write x Write x

AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants, ASPLOS'06  
Associating synchronization constraints with data in an object-oriented language, POPL'06

### Solution to problem 2

- Atomicity violation = unserializable interleaving

Thread 1

```

access x
atomic ?
access x
                    
```

Thread 2

```

-accesses
                    
```

Read x

```

atomic
Read x
                    
```

Write x

```

not atomic
Read x
Write x
                    
```

- 4 out of 8 cases are violations

<p>Read x Write x Read x</p> <p>Inconsistent views</p>	<p>Write x Write x Read x</p> <p>Too early overwritten</p>	<p>Write x Write x Read x</p> <p>Leaking intermediate value</p>	<p>Read x Write x Write x</p> <p>Using stale value</p>
--	--	---	--

Both hardware and software solutions exist

45

### Solution to problem 1

- Which code regions are expected to be atomic?
  - Manual annotation
  - ??

Thread 1

```

if (proc){
tmp=*proc;
}
MySQL
                    
```

Thread 1

```

while (!flag) {}
                    
```


Thread 2

```

flag=TRUE;
                    
```

AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants, ASPLOS'06

### Inference based bug detection



48




### Infer likely program invariants

- What is the typical value of x?
- What is the ...?
- How to use it to detect general semantic bugs?
- How to use it to detect memory bugs?
- How to use it to detect concurrency bugs?

### Solution to problem 1

- Which code regions are expected to be atomic?
  - Manual annotation
  - Training/Learning
  - Testing validation



Thread 1      Thread 2

```

if (proc){
  tmp=*proc;
}
MySQL
                    
```

Thread 1      Thread 2

```

while (!flag) {};   flag=TRUE;
                    
```

AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants, ASPLOS'06

### What are order violations?

- Expected order between two operations are flipped
- Can it be detected by atom. vio. detectors?
- Can it be detected by race detectors?

Thread 1 (child)      Thread 2 (parent)

```

Thread 1:
_state = mThd->state;
                    
```

↙      ↘

```

Thread 2:
mThd#CreateThd();
                    
```

↘      ↙

Mozilla

51

### How to detect order violation?

- Problem 1
  - How to judge which is the correct order?
- Problem 2
  - How to detect the order violation?

Thread 1 (child)      Thread 2 (parent)

```

Thread 1:
_state = mThd->state;
                    
```

↙      ↘

```

Thread 2:
mThd#CreateThd();
                    
```

↘      ↙

Mozilla

52

### Solutions

- How to judge which is the correct order?
  - Learning based techniques [Micro'09, OOPSLA'10]
  - Semantic guided techniques [ASPLOS'11]
- How to detect the order violation
  - Easy

Thread 1 (child)      Thread 2 (parent)

```

Thread 1:
_state = mThd->state;
                    
```

↙      ↘

```

Thread 2:
mThd#CreateThd();
                    
```

↘      ↙

Mozilla

53

### What are multi-var conc. bugs?

- Multi-variable bugs
  - Untimely accesses to correlated variables
- Can it be detected by race detectors?
- Can it be detected by AVIO?

Thread 1      Thread 2

```

Thread 1:
InProgress=FALSE;
URL = NULL;
                    
```

↙      ↘

```

Thread 2:
if(InProgress)
  isBusy=TRUE;
                    
```

```

                    if(isBusy) {
                      if(URL == NULL)
                        __assert_fail(),
                      ...
                    }
                    
```

Mozilla

54

### How to detect multi-variable bugs?

- Problem 1
  - How to judge which variables are correlated?
- Problem 2
  - How to detect untimely accesses

```

Thread 1          Thread 2
InProgress=FALSE; if(!InProgress)
URL = NULL;       isBusy=TRUE;
                  if(isBusy) {
                  if(URL == NULL)
                  ..._assert_fail(),
                  ...
            
```

*Mozilla*

55

### Solutions

- Which variables are correlated?
  - Variables that are frequently accessed together
- How to detect the violation?
  - Extend existing single-variable bug detectors

```

struct JSCache {
...
JSEntry table[SIZE];
bool empty;
...
}
            
```

*Mozilla*

```

struct JSRuntime {
...
int totalString;
double lengthSum;
...
}
            
```

*Mozilla*

```

struct fb_var_screeninfo
{
...
int red_msb;
int blue_msb;
int green_msb;
int transp_msb;
}
            
```

*Linux*

MUVI.SOSP'07, ColorSafe.ISCA10

### Solutions

- Which variables are correlated?
  - Variables that are frequently accessed together
- How to detect the violation?
  - Extend existing single-variable bug detectors

```

Thread 1          Thread 2
InProgress=FALSE; if(!InProgress)
URL = NULL;       isBusy=TRUE;
                  if(isBusy) {
                  if(URL == NULL)
                  ..._assert_fail(),
                  ...
            
```

*Mozilla*

MUVI.SOSP'07, ColorSafe.ISCA10

### Are we done?

- Are these “learning”-based techniques perfect?

58

### Are we done?

- False positives
  - Still a problem!
- False negatives
  - Still a problem!

59

### Break


60

### How to do better?

```

Thread 1      Thread 2
if (proc) {   → proc = NULL;
  tmp = *proc;
}
    
```

MySQL



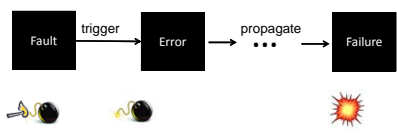
61

### How to do better?

If we cannot find a more accurate **root-cause** pattern, let's look at the **effect** patterns of concurrency bugs!

62

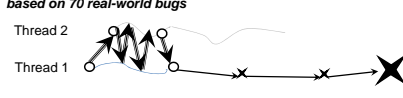
### The lifecycle of bugs



63

### The lifecycle of (most) concurrency bugs

*based on 70 real-world bugs*



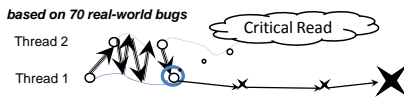
Fault → trigger

- Data races
- Atomicity violations
- single variable
- multiple variables
- Order violations
- ...

64

### The lifecycle of (most) concurrency bugs

*based on 70 real-world bugs*



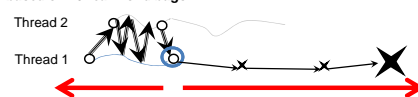
Error → propagate

- Memory errors
  - NULL ptr
  - Dangling ptr
  - Uninitialized read
  - Buffer overflow
- Semantic errors

65

### The lifecycle of (most) concurrency bugs

*based on 70 real-world bugs*



Fault → trigger → Error → propagate → Failure

short single-threaded

- Crash @ invalid memory
- Crash @ assertion
- Infinite loops
- Incorrect outputs
- Error messages

66

### Examples

Thread 1      Thread 2

```

if (proc){
  tmp = *proc;
}
        
```

MySQL

Thread 1      Thread 2

```

mThd = CreateThd();
_state = mThd->state;
        
```

Mozilla

atomicity violation → null\_ptr deref. → crash

order violation → uninitialized read → crash

67

### Summary of effect characteristics

- Simple error/failure patterns
- Single-threaded error propagation
- Short error propagation

68

### Cause-oriented approach

Thread 1   Thread 2  
 read x    write x

Thread 1   Thread 2  
 read x    write x

- Limitations
  - False positives
  - False negatives

69

### Effect-oriented approach

- Step 1: Statically identify **potential failure/error site**
- Step 2: Statically look for **critical reads**
- Step 3: Dynamically identify **buggy interleaving**

Fewer false positive  
Fewer false negative

70

### Our tools

<ul style="list-style-type: none"> <li>• Memory errors</li> <li>• NULL ptr</li> <li>• Dangling ptr</li> <li>• Uninitialized read</li> <li>• Buffer overflow</li> <li>• Semantic errors</li> </ul>	<ul style="list-style-type: none"> <li>• Crash @ invalid memory</li> <li>• Crash @ assertion</li> <li>• Infinite loops</li> <li>• Incorrect outputs</li> <li>• Error messages</li> </ul>
---	--

ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach, ASPLOS'10<sup>1</sup>

### Our tools

<ul style="list-style-type: none"> <li>• Memory errors</li> <li>• NULL ptr</li> <li>• Dangling ptr</li> <li>• Uninitialized read</li> <li>• Buffer overflow</li> <li>• Semantic errors</li> </ul>	<ul style="list-style-type: none"> <li>• Crash @ invalid memory</li> <li>• Crash @ assertion</li> <li>• Infinite loops</li> <li>• Incorrect outputs</li> <li>• Error messages</li> </ul>
---	--

ConSeq: Detecting Concurrency Bugs through Sequential Errors, ASPLOS'11<sup>2</sup>

## Slide 70

---

**SL29** Shan Lu, 2014-1-7

**SL30** i like the mapping in paper: cause maps to xxx effects; effect map back to xxx.  
Shan Lu, 2014-1-7

**SL31** if i refer to interleaving here, we need to define interleaving earlier  
Shan Lu, 2014-1-8

### Our tools

Thread 2

Thread 1

Fault → trigger → Error → propagate → Failure

- Crash @ assertion
- Infinite loops
- Incorrect outputs
- Error messages

• Semantic errors

*ConSeq: Detecting Concurrency Bugs through Sequential Errors, ASPLOS'11 73*

### ConSeq bug example

```

Thread 1      Thread 2
InProgress=FALSE;  if(!InProgress)
URL = NULL;      isBusy=TRUE;
                  if(!isBusy) {
                    if(URL == NULL)
                      __assert_fail(),
                    ...
                  }
Mozillo
    
```

MJVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. Shan Lu, et al., SOSPO9 74

### Step 1: Identify potential failure sites

Statically look for places where failures could happen

Failure Type
Assertion Failure
Error Message
Incorrect output
Infinite loop

Number of failure sites in MySQL: ~1000

75

### Step 1: Identify potential failure sites

Statically look for places where failures could happen

```

if(!InProgress)
  isBusy=TRUE;
if(!isBusy){
  if(URL == NULL){
    __assert_fail();
  }
}
    
```

76

### Step 2: Look for critical reads

Statically find shared mem. reads that impact failure sites

```

if(!InProgress)
  isBusy=TRUE;
if(!isBusy){
  if(URL == NULL){
    __assert_fail();
  }
}
    
```

Static slicing

77

### Stage 3: Look for buggy interleavings

Dynamic analysis looks for interleavings that provide critical reads with bad values

```

Thread 1      Thread 2
...
InProgress=FALSE;  if(!InProgress)
URL = NULL;      isBusy=TRUE;
                  if(!isBusy) {
                    if(URL == NULL){
                      __assert_fail();
                    }
                  }
    
```

78

## Slide 74

---

**SL32** the sosp, muvi reference should be put earlier  
Shan Lu, 2014-1-8

### Look for alternative data dependence

THD	R/W	Addr	Value
1	W	0xabcd	✓
1	R	0xabcd	✓
2	W	0xabcd	✗

*Is the alternative data dependence feasible in future runs?*

### Dependence feasibility analysis

- Can synchronization prevent a data dependence?

### Dependence feasibility analysis

- Locks could make a data-dependence infeasible

- Barriers could make a data-dependence infeasible

### Put everything together

```

    Identify failure sites → Identify Critical Reads → Identify Suspect Interleavings → Suspect interleaving Testing → Bug reports
  
```

```

    Thread 1: InProgress=FALSE; URL=NULL;
    Thread 2: if(isBusy=TRUE; if(isBusy){ if(proc == NULL){ assert_fail(); } }
  
```

### ConMem

```

    Fault → trigger → Error → propagate → Failure
  
```

- Memory errors
  - NULL ptr
  - Dangling ptr
  - Uninitialized read
  - Buffer overflow
- Semantic errors
- Crash @ invalid memory
  - Crash @ assertion
  - Infinite loops
  - Incorrect outputs
  - Error messages

*ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach, ASPLOS'10*

### ConMem bug example

- What are the errors?
- How to detect them using dynamic analysis?

```

    Thread 1: if(proc){ tmp=*proc; } MySQL
    Thread 2: proc = NULL; mThd=CreateThd(); _state = mThd->state; Mozilla
  
```



## 5-min Break?

85

## Summary of conc. bug detection

- How to detect them?
  - Find patterns
    - Cause patterns
    - Effect patterns
- What are the remaining challenges?
  - Performance
  - False negatives [@Enact,ISCA03, ParaLog,ASPLOS10, RaceMob,SOSP13, LiteRace, ...]
  - False positives
    - Customized synchronization
- The state of practice
  - Race detection; Atom. detection; ...

86

## Outline

- What are concurrency bugs
- Concurrency bug detection
- Concurrency bug exposing
- Concurrency bug failure recovery
- Concurrency bug fixing
- Others
- Conclusion

87

## Exposing Concurrency Bugs

88

## Background --- Software Testing

- Testing space
- Coverage criteria
  - Testing property
- Test suite
- Software testing is extremely important!

89

## The challenges

- Huge state space
- What is the coverage criteria?
- How to cover a testing property?

90

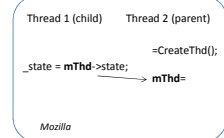
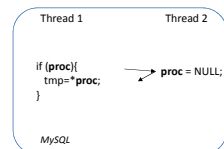
## Background in testing

- Coverage criteria
  - Examples
  - Complexity vs. Capability
- Test input design

91

## What are the coverage criteria?

- Total-order [TSE92]
- ALL-DU [ICSM92, ISSTA98]
- Synchronization [PPoPP05]
- Function [SoQua07]
- Bug-pattern based  
[Chess, RaceFuzzer, CTrigger...]



92

## How to cover a testing property?

How can I make A execute before B?

- Ad-hoc solution
  - Single-core based
  - Multi-core based
- Constraint-solving based solution [Madhu Viswanathan, NEC]
- How many properties can be covered in one run?  
[Madan Musuvathi]

93

## Summary of exposing con. bugs

- Key challenges
- Key solutions
- What are the remaining challenges?
  - Better coverage criteria
  - Input generation
  - Regression testing
  - Unit testing

94

## Summary of the day

- Concurrency bug detection
  - Cause based detection
    - Data race; atomicity violation; order violation; single variable; multi-variable
  - Effect based detection
  - Bug exposing (testing)
- Detection mechanisms
  - Run-time analysis
  - Static analysis
  - Learning-based technique

95