Fixing concurrency bugs

Shan Lu

1

How to automatically fix bugs?

- How to automatically fix memory bugs?
- How to automatically fix semantic bugs?

2

How to fix these bugs?

```
Thread 1                 Thread 2

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

MySQL
```

```
Thread 1 (parent)              Thread 2 (child)

printf("%u\n", End);           End = time();
printf("%u\n", End-start);

FFT
```

3

Opportunities

- Concurrency bugs are easier to fix automatically!
  – How to fix an atomicity violation? lock
  – How to fix an order violation? Signal/wait

4

Challenges

- Q1: what is the root cause?
  – Atomicity violation? Order violation? A mix of both?
- Q2: how to enforce a specific synchronization?
  – Do not introduce new bugs
    - How could a patch introduce new bugs?
  – Do not hurt performance too much
    - How could a patch hurt performance?
  – Do not hurt code readability too much
    - How could a patch hurt readability?

5

Q1. what is the root cause?

- Leverage automated bug detection tools
- What is the root cause of a data race?

```
Thread 1                 Thread 2

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

MySQL
```

```
Thread 1 (parent)              Thread 2 (child)

printf("%u\n", End);           End = time();
printf("%u\n", End-start);

FFT
```

6

Q1. what is the root cause?

- Leverage automated bug detection tools
- What is the root cause of a bug detected by an atomicity-violation detector?

```
Thread 1                Thread 2

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

MySQL
```

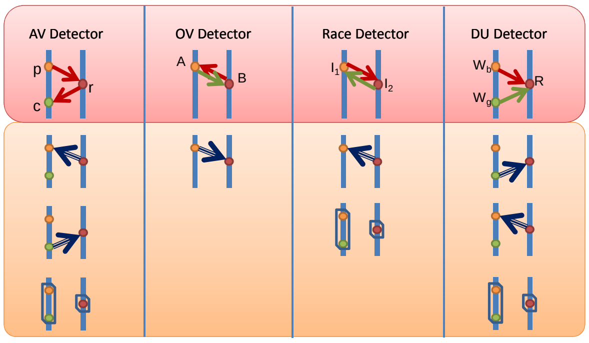
```
Thread 1 (parent)                Thread 2 (child)

printf("%u\n", End);             End = time();
printf("%u\n", End-start);

FFT
```

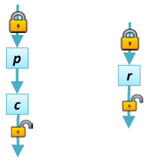
Fix strategy design

Preparing for inaccurate bug-detection results



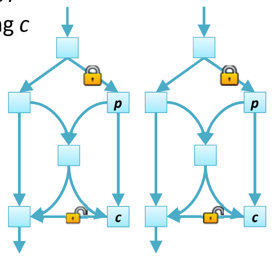
Q2.a how to enforce atomicity?

- How to make *p-c* code region mutually exclusive with *r*
  - Put *p* and *c* into a critical section
  - Put *r* into a critical section
  - Select or introduce a lock for the two critical sections



Potential problems

- A naïve solution
  - Add lock on edges reaching *p*
  - Add unlock on edges leaving *c*
- Potential new bugs
  - Could lock without unlock
  - Could unlock without lock
  - etc.
- Simpler examples …



Solutions?

- How to fix these?

```
if (..){          while(..){p};
  p
}
c                 c;
```

Solutions?

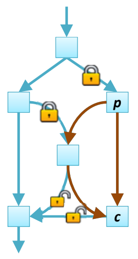
- How to fix these?

```
if (..){          while(..){p};
  p
}
c                 c;
```

How to generalize this into an algorithm?

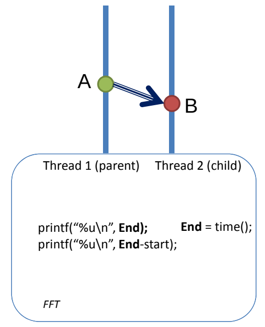
Solutions

- Step 1: find protected nodes in critical section
  - In *f*'s CFG, find nodes on any *p* → *c* path
- Step 2: add lock operations
  - unprotected node → protected node
  - protected node → unprotected node
- Avoid those potential bugs mentioned



Q2.b how to enforce order?

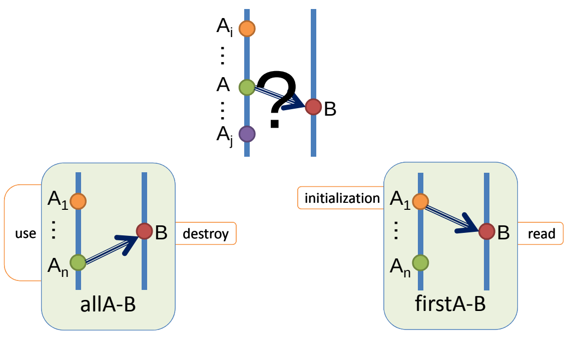
- How to make instruction A execute before B?



```
printf("%u\n", End);          End = time();
printf("%u\n", End-start);
```

Automated Concurrency-Bug Fixing, OSDI'12

What if A has multiple instances?



Challenges for AllA-B

- Does signal after A, wait before B work?

Thread 1 Thread 2

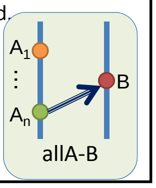
```
*P=tmp;

              free(P);
```

- What if A is executed for multiple times in its thread?
- What if there are multiple instances of thread-A?

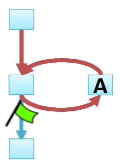
Solutions for AllA-B (principles)

- *signal* in A-threads:
  - A-thread signals when it will not execute more A;
  - Each A-thread signals only once;
  - Each A-thread signals as soon as possible.
- *wait* before B:
  - B Proceeds when each A-thread has signaled.



Solutions for AllA-B (A-side)

```
. . .;
for (. . .)
    . . . ; // A
. . .;
```

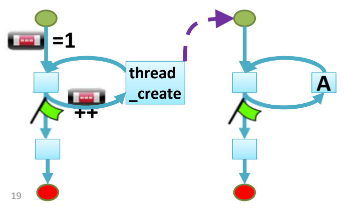


- Each thread that execute A:
  - exactly once as soon as it can execute no more A.

Solutions for AllA-B (A-side)

```
void main() {
 for (. . .)
  thread_create(thr_main);
 . . .;
}
```

```
void thr_main() {
 for (. . .)
  . . . ; // A
 . . .;
}
```

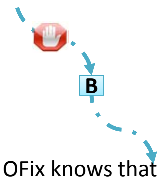


counter for signal threads

```
void ofix_signal() {
 mutex_lock(L);
 -;
 if ( = 0)
  cond_broadcast(con);
 mutex_unlock(L);
}
```

Solutions for AllA-B (B-side)

- Safe to execute only when is 0.



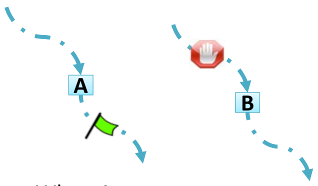
```
void ofix_wait() {
 mutex_lock(L);
 if ( = 0)
  cond_timedwait(con, L, t);
 mutex_unlock(L);
}
```

- Give up if OFix knows that it introduces new deadlock.
- Timed wait-operation to mask potential deadlocks.

Solutions for FirstA-B?

Solutions for FirstA-B

- Basic enforcement



```
void ofix_signal_b() {
 if (flag != true) {
  flag = true;
  mutex_lock(L);
  cond_broadcast(con);
  mutex_unlock(L);
 }
}
```

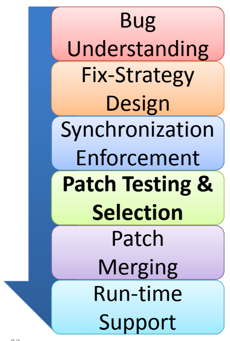
```
void ofix_wait_b() {
 mutex_lock(L);
 if (flag != true)
  cond_timedwait(con, L, t);
 mutex_unlock(L);
}
```

- When A may not execute:
  - Add a safety-net of signal with allA-B algorithm.

Is that all?

- Is the patch really correct?
  - Could it lead to bugs?
- What is the readability?
- What is the performance?

The whole tool chain



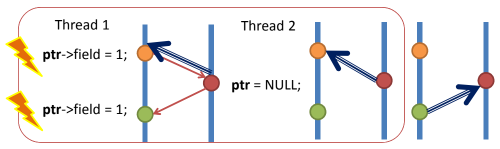
Patch Testing

- Prune incorrect patches:
  - Patches causing failures due to incorrect root causes, etc.
- Prune slow patches
- Prune complicated patches

- Not exhaustive testing, but patch oriented testing.

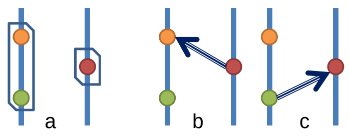
Run Once without External Perturbation

- Reject if there is a time-out or failure.
- Patches fixing wrong root cause:
  - Make software to fail deterministically.



Implicit Bad Patch

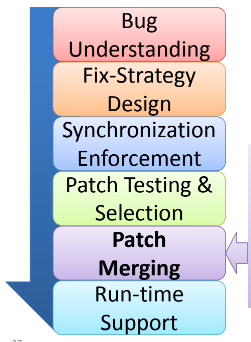
- A failure in patch_b implies a failure in patch_a:
  - If patch_a is less restrictive than patch_b.



Mutual Exclusion Order Relationships

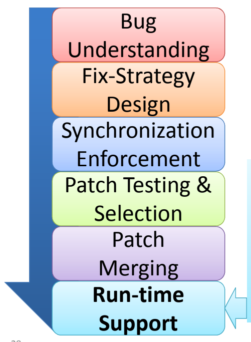
- Helpful to prune patch_a:
  - Traditional testing may not find the failure in patch_a.

Patch Merging



- One programming mistake usually leads to multiple bugs.
- Heuristics to merge patches for related bugs.

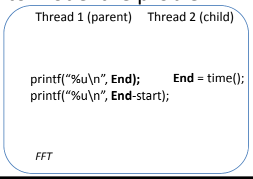
CFix: Automated Concurrency-Bug Fixing



- To understand whether there is a deadlock underlying time-out.
- Low-overhead, and suitable for production runs.

Summary

- Key challenges
- Key solutions
- Remaining challenges
  - Handle more complicated bugs
  - Learning from human patches
  - Other way to model the problem



Break

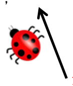
What is the remaining problem?

An bug example

```
// child thread          // parent thread
fputs(fp, ...);



                         fp = NULL;
```



How would you fix this bug?

Manual patch vs. Auto Match

```
// child thread          // parent thread
fputs(fp, ...);          + lock(L);
+ lock(L);               + while (cnt > 0)
+ signal(cond);          +  wait(cond, L);
+ cnt--;                 + unlock(L);
+ unlock(L);             fp = NULL;
```

```
// child thread       // parent thread
fputs(fp, ...);


                      + thread_join(...);
                      fp = NULL;
```

Auto Patch could be much more complicated

One more example

```
//child thread                //parent thread
if (…) {                      FIFO= NULL;
  unlock(FIFO->m);
  return;
}
```

How many signals do we need to fix this bug?

How do developers fix con-bugs?

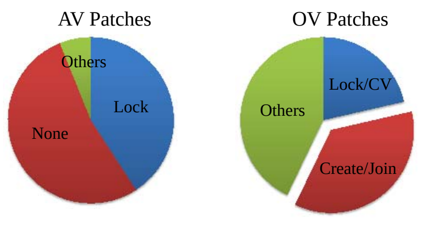
- How can we find this out?

Empirical study -- methodology

37

Empirical study – finding 1

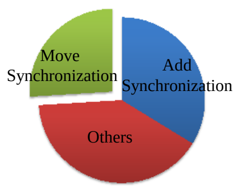
- What synchronization primitives are used?



Synchronization primitives in Patches

38

Empirical study – finding 2



Fix strategies in Patches

39

What can be automated?

40

What can be automated?

- Adding join
- Fixing bugs by code moving

41

When does Add-Join work?

42

When does Add-Join work?

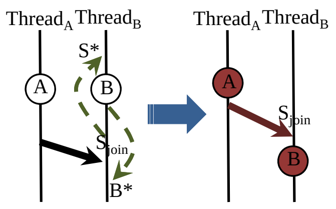
- Parent-child relationship
- Not-joined yet
- Joinable child thread
- No deadlock risk

How to add join

When can Move help?

- When can Move help fix an OV bug?
- When can Move help fix an AV bug?

Moving to fix OV bugs



Movejoin fixes OV bugs
move-up S or move-down B

Move-create can also help

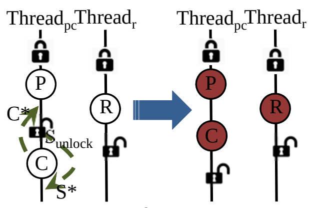
An example of move-create

```
  //parent thread
  void tr_sessionInit (...) {
    h = malloc(...);
+   h->band = bdNew(h);
    tr_eventInit(...);
    ...
-   h->band = bdNew(h); //A
  }

  void tr_eventInit (...) {
    pthread_create(...);
  }
```

```
//child thread
assert(h->band); //B
```

Moving to fix AV bugs



Moveunlock fixes AV bugs
move-up C or move-down S

What are the challenges?

- Data dependency checking (static)
  - What could be wrong?
- Control dependency checking (static)
  - What could be wrong?

49

Other research in this field

Demmunix.OSDI08

Avisio.ASPLOS13

Grail.FSE15, Gadara.OSDI08

50

Summary

- Constraints in automated bug fixing
  - Correctness
  - Performance
  - Readability
- Concurrency bugs can be automatically fixed
- Different ways to fix concurrency bugs
  - Adding synchronization
    - Lock, C.V., join
  - Moving memory accesses and synchronization around

51

Break

52

Many other things

- Deterministic execution/program
- Record-and-replay
- Model checking & symbolic execution
- Approximate computation

53