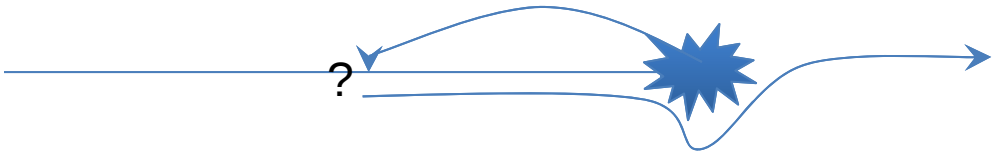
Recovering From Concurrency-bug Failures

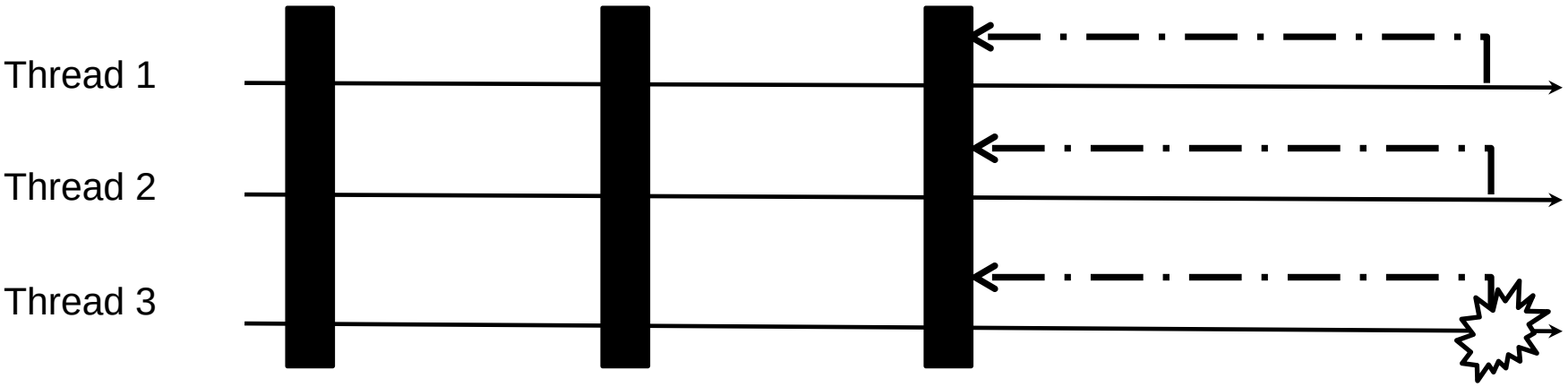
The challenges for failure recovery

- What is a correct program state?
- How to go back to that state?
- How to by-pass the failure during re-execution?



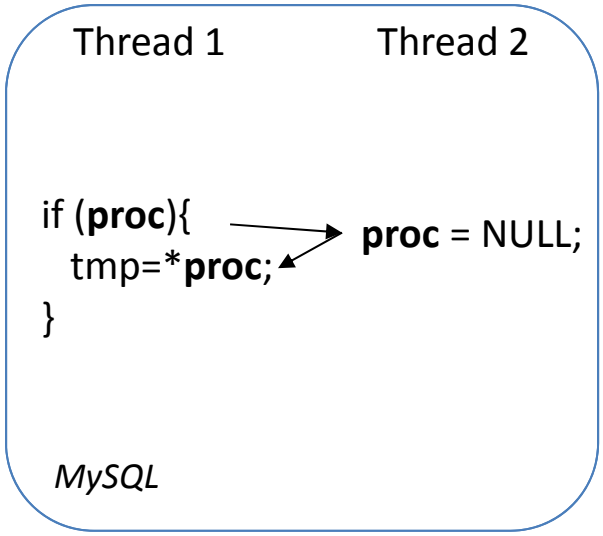
Traditional rollback-recovery

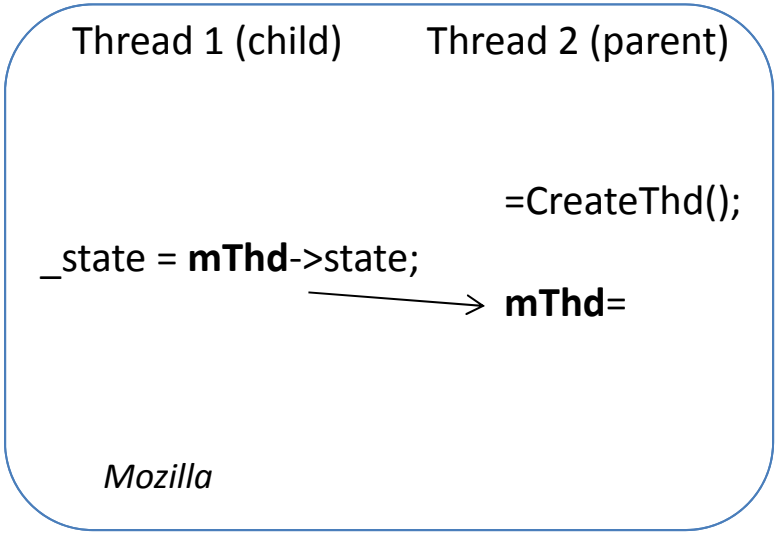
Concurrency bugs are easier to recover from!



Triage: Diagnosing Production Run Failures at the User's Site, SOSP'07

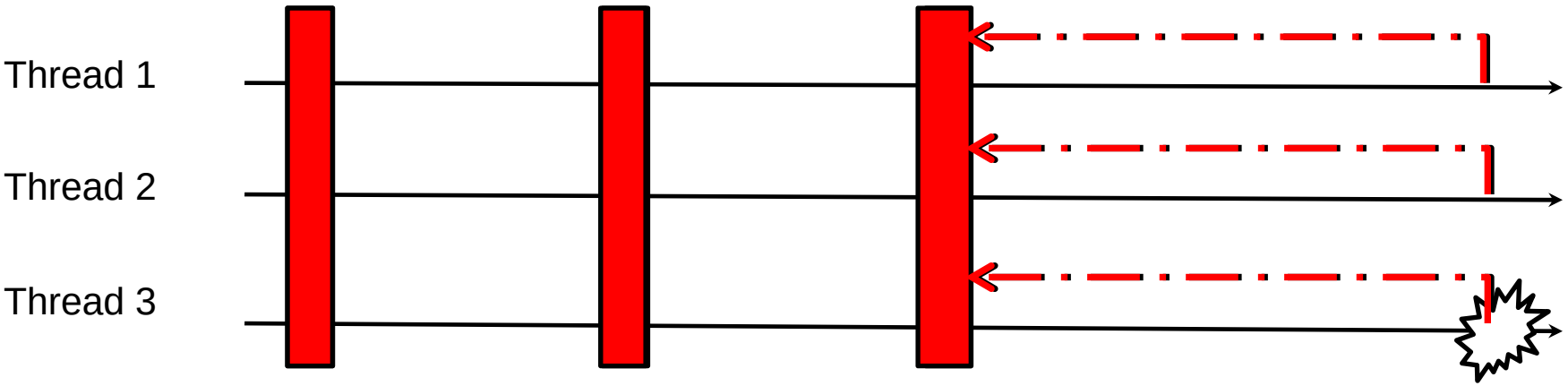
Examples


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


Thread 1 (child)
Thread 2 (parent)
=CreateThd();
_state = mThd->state;
mThd=
Mozilla

Are we done?

- What about the performance?



Modify OS/HW and Slow

How can we do better?

Do we have to roll back all threads?

Do we have to make whole-memory checkpoint?

ConAir: Featherweight Concurrency Bug Recovery Via Single-Threaded Idempotent Execution. ASPLOS'13

Bug example revisit

Do we need to roll back all threads?

Thd1	Thd2
if(ptr){	
	ptr = NULL;
tmp = *ptr;	
}	

Bug example revisit

Thd1

```
if(ptr){

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

Thd2

```
ptr = NULL;
```

Timeline

Rollback one thread is enough

Bug example revisit

Do we need periodic checkpoint?

Thd1	Thd2
if(ptr){	
	ptr = NULL;
tmp = *ptr;	
}	

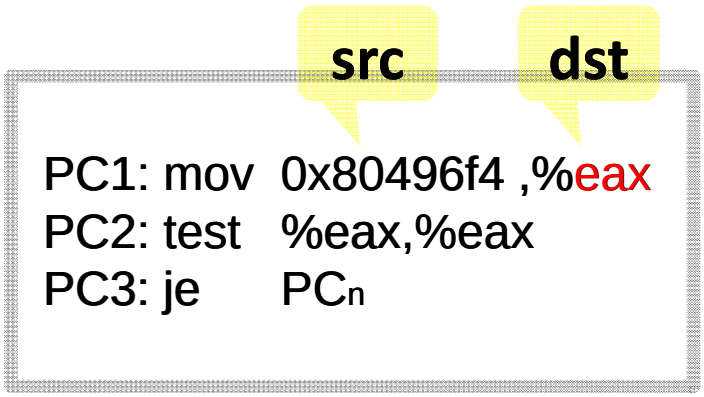
Failure site is (somewhat) predictable, guided checkpoint could work

Bug example revisit

Do we need memory checkpoint?

if(ptr)

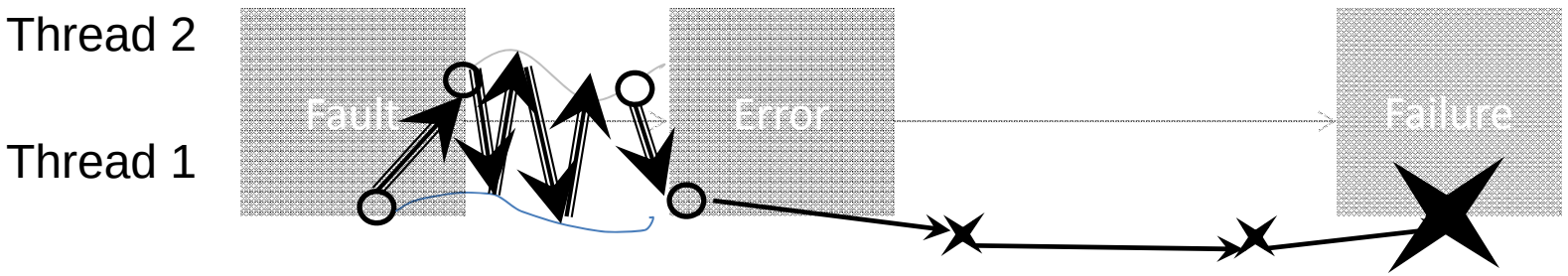
Idempotent



An idempotent region is a code region that can be reexecuted for any number of times without changing the program semantics

Re-execution region is small, no need to checkpoint

Generalize the bug example



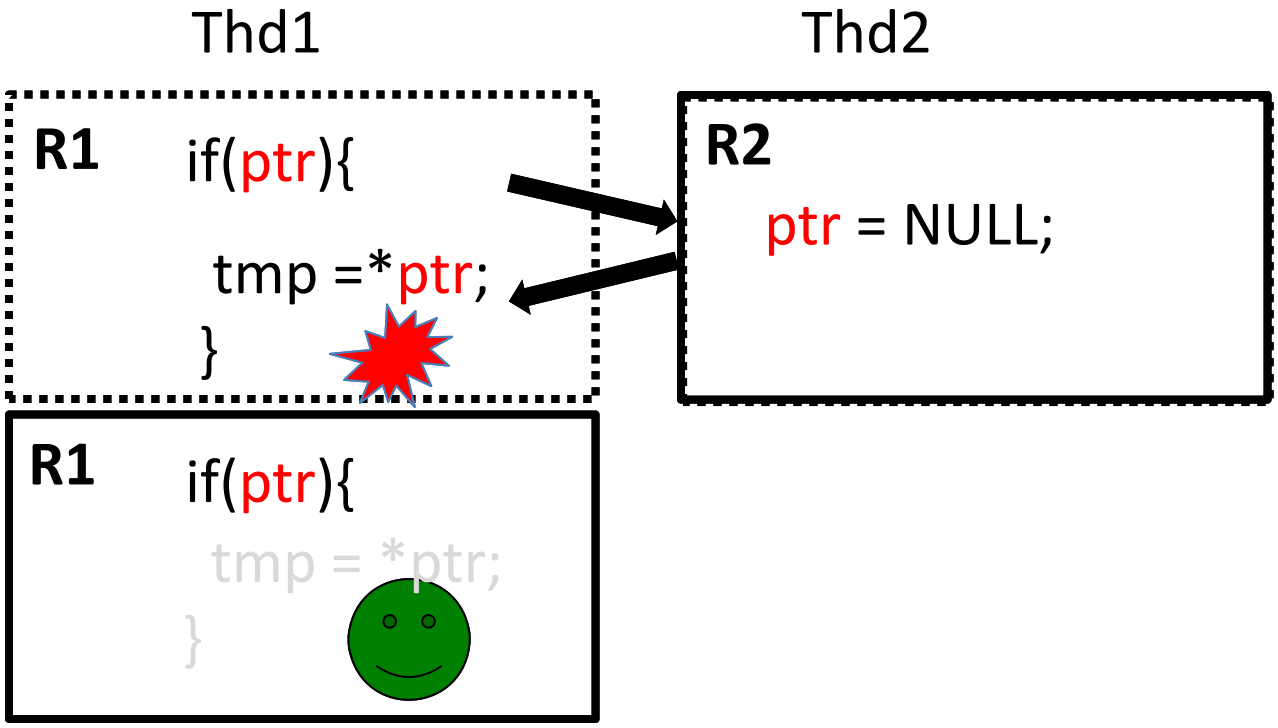
Single-threaded error propagation → roll back the failure thread is mostly enough

Simple failure/error patterns → guided checkpoint-recovery is mostly enough

Short error propagation → re-executing an idempotent region is often enough

Is single-threaded re-exec enough?

- Atomicity violation bugs



Is single-threaded re-exec enough?

- Atomicity violation bugs (~100%)
- Order violation bugs

Thd1	Thd2
_state = mThd->state;	
	mThd = CreateThd();

Is single-threaded re-exec enough?

- Atomicity violation bugs (~100%)
- Order violation bugs

Thd1

Thd2

_state = mThd->state;

_state = mThd->state;

_state = mThd->state;

mThd = CreateThd();

Failure thread executes too fast

Failure thread executes too slow

Is single-threaded re-exec enough?

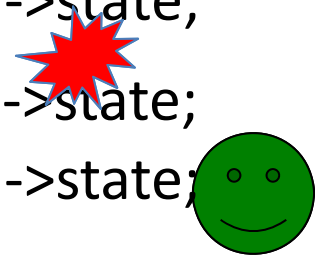
- Atomicity violation bugs (~100%)
- Order violation bugs (50%)

Thd1

Thd2

```
_state = mThd->state;
_state = mThd->state;
_state = mThd->state;
```

```
mThd = CreateThd();
```



Failure thread executes too fast

Failure thread executes too slow

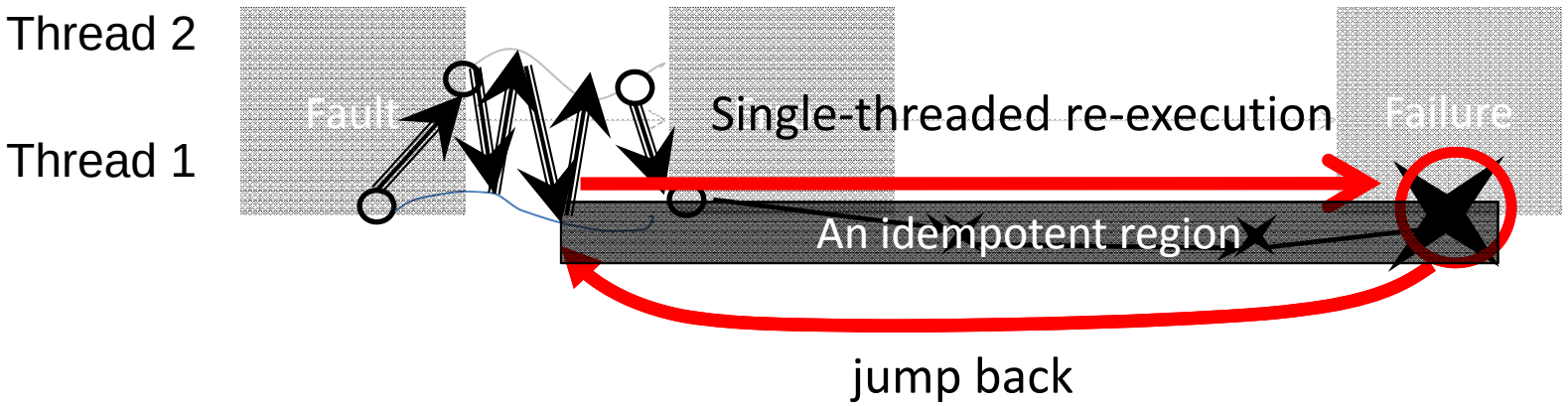
Is single-threaded re-exec enough?

- Atomicity violation bugs (~100%)
- Order violation bugs (50%)
- Deadlocks (?)

Is single-threaded re-exec enough?

- Atomicity violation bugs (~100%)
- Order violation bugs (50%)
- Deadlocks (100%)

A simplified con. bug failure recovery



Step1: Locate potential failures

Step2: Identify the re-exec region

Step3: Generate rollback re-exec code

Recover many failures
Negligible cost
No change to semantics

Step 1 Identify potential failure sites

Failure Type	Potential Failure Site
Assertion Failures	Call to __assert_fail etc
Error Messages	Call to fprintf(stderr,…), NS_WARNING in Mozilla, tr_err in Transmission, etc.
Incorrect outputs	Call to (f)printf, BinLog::Write in MySQL, etc.
Illegal mem. accesses	Dereferencing
Deadlocks	Call to pthread_mutex_lock(…)

Number of potential failure sites in MySQL: ~13000

Step 2 identify re-execution region

- Re-execution region = idempotent region

What code region is idempotent?

What makes code non-idempotent ?

- I/O operation 
- Shared memory write 
- Local memory writes

```
X = X + 1;
Z = X + Y;
```

```
Y = X + 1;
Z = X + Y;
```

- Writes to registers

What makes code non-idempotent ?

- I/O operation

CallInst(to lib)

- Shared memory write

StoreInst to non-Alloca

- Local memory writes

X = X + 1;
Z = X + Y;

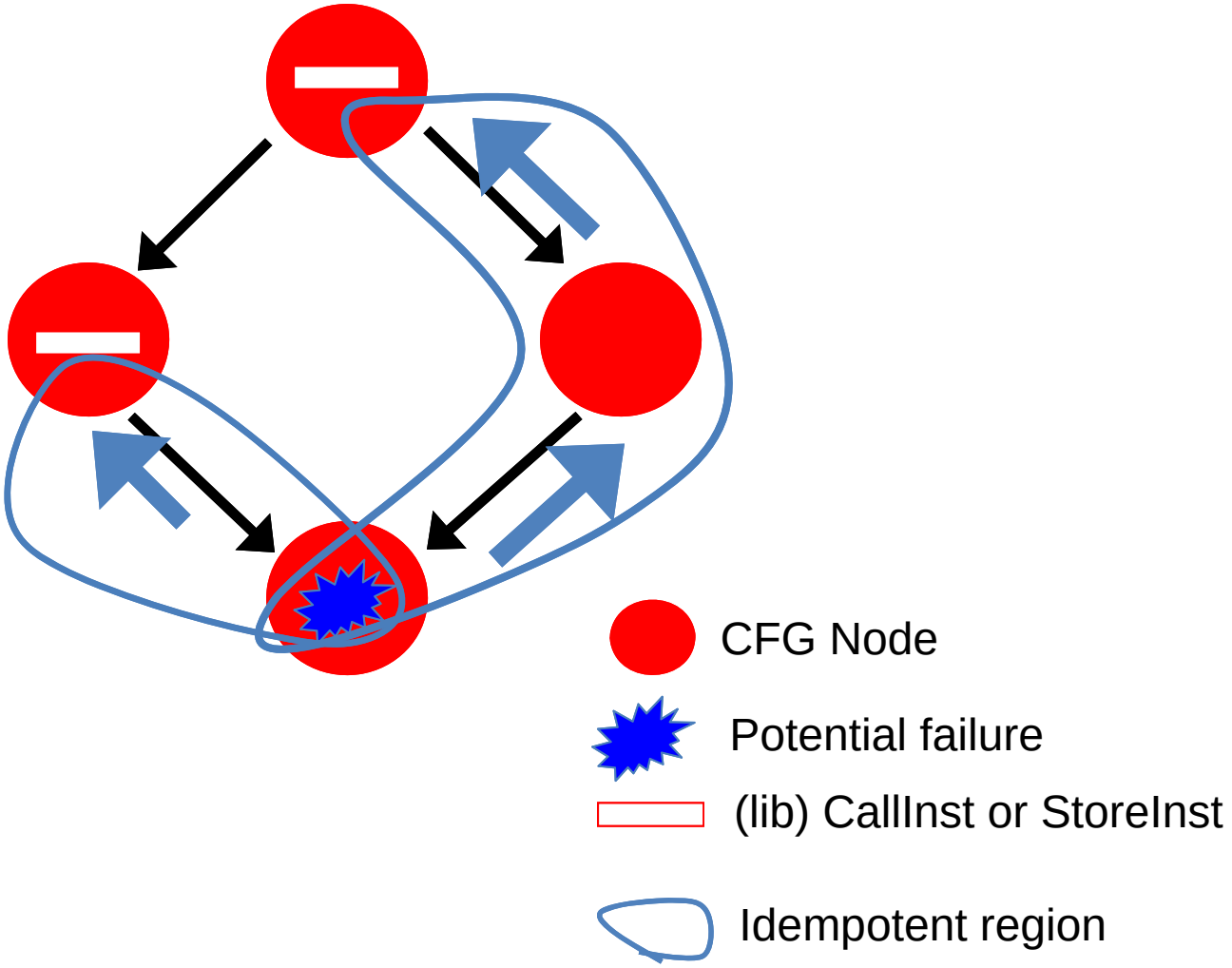
StoreInst to Alloca

Y = X + 1;
Z = X + Y;

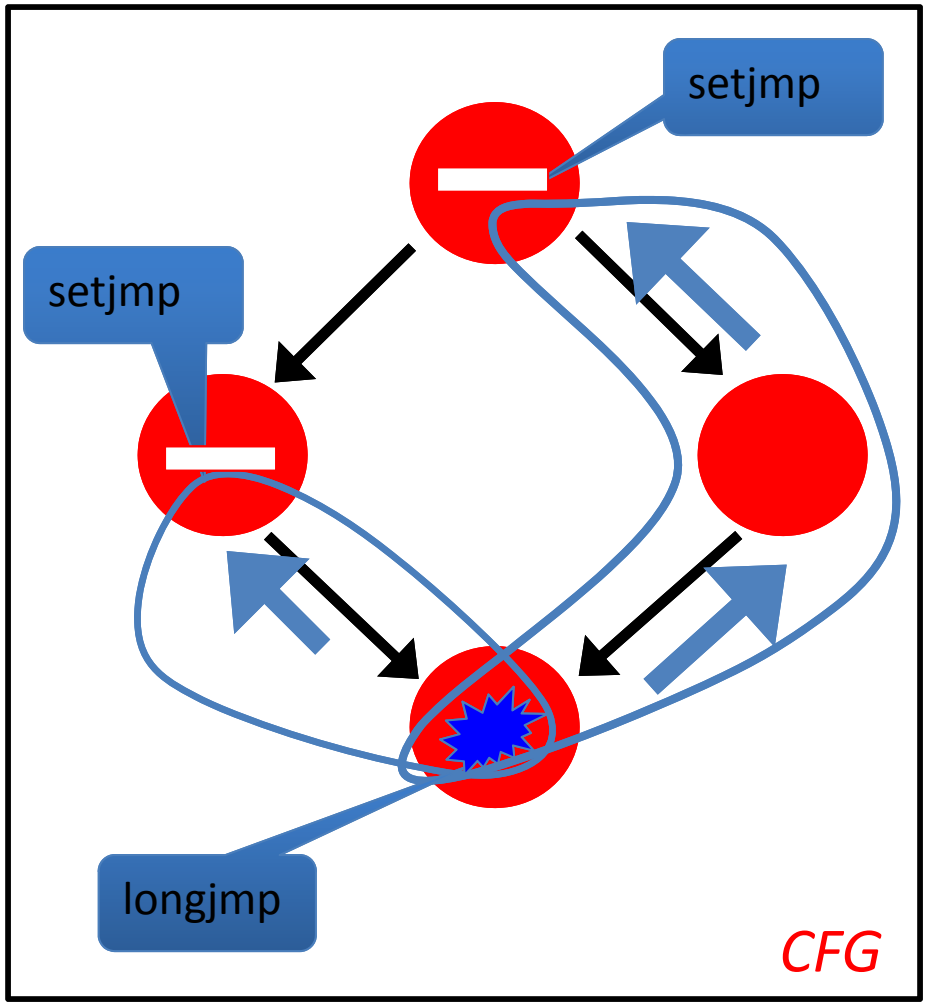
Write to virtual register
+ setjmp/longjmp

- Writes to registers

Step 2 identify re-execution region



Step 3 generate code




```
setjmp(...);
//start of idempotent region
…
if(e){
}else{
    while(retryCnt++ < MAX){
      longjmp(...);
    }
   //potential failure site
   _assert_fail();
}
```

Code

Maximize legit idempotent region

- Handle some library functions (e.g. malloc, lock)
  - During execution: timestamp
  - Upon failure: undo most recent library functions
- Inter-procedural analysis
  - Configurable max level of function calls (e.g. 3)
- Optimization
  - Some recovery attempts are doomed to fail

Summary for failure recovery

- How to recovery from concurrency bug failures?
  - Rollback-replay
    - Different types of replay …
- What are the remaining challenges?
  - Coverage vs. Overhead/System-Support
  - Can we prevent failures at run time?

Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. OSDI 2008

Cooperative Empirical Failure Avoidance for Multithreaded Programs. ASPLOS13

Failure Prevention

"AI: a Lightweight System for Tolerating Concurrency Bugs"

Mingxing Zhang, Yongwei Wu, Shan Lu, Shanxiang Qi, Jinglei Ren, Weimin Zheng, FSE 2014

What is con. bug failure prevention?

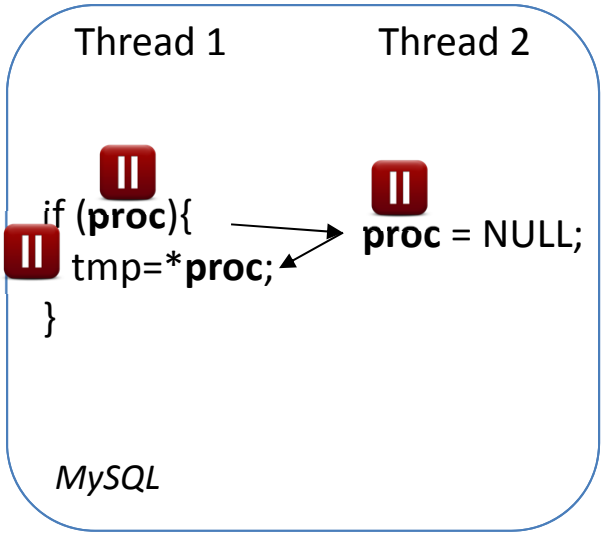
- How to predict a failure?
- How to change the execution and avoid the failure?
  - Pause to change the timing!

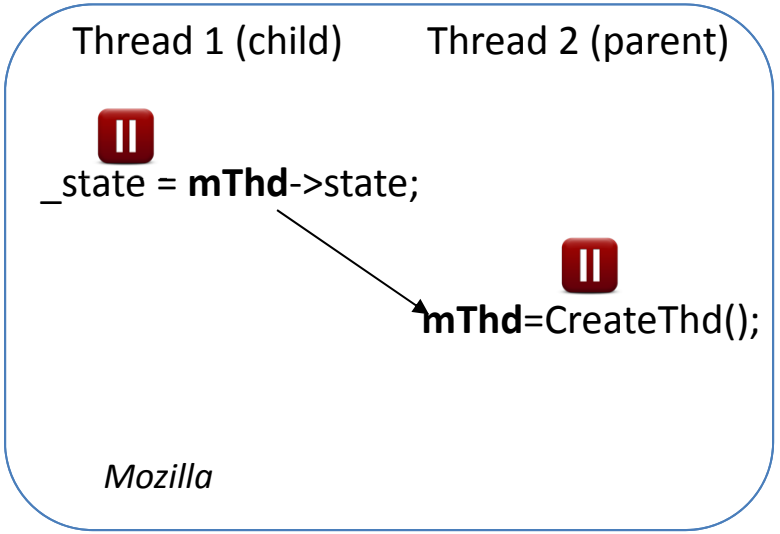
Challenge

- How to predict a failure?
  - Not too early
    - Too early will lead to unnecessary performance losses
  - Not too late
    - Too late will make failures inevitable

Example

- Where should we predict the failure and pause?





How to generalize?

- Stop before every shared-variable write?

- Stop before every shared-variable read?

How to generalize?

- Stop before every shared-variable write?
  - When the previous access is abnormal?
- Stop before every shared-variable read?
  - When the previous access is abnormal?

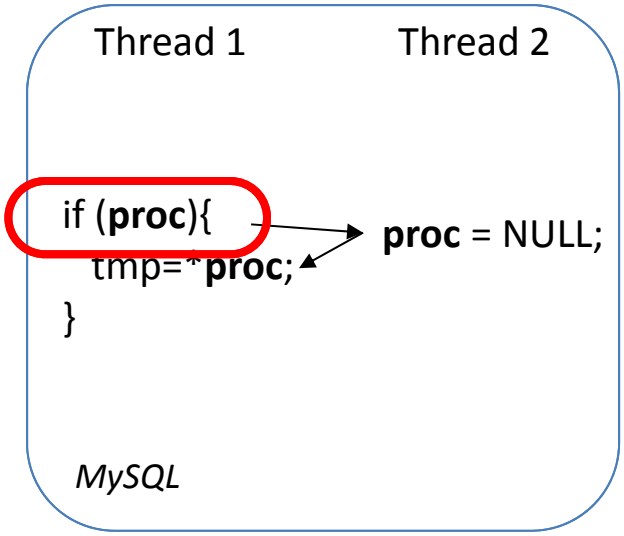
How to generalize?

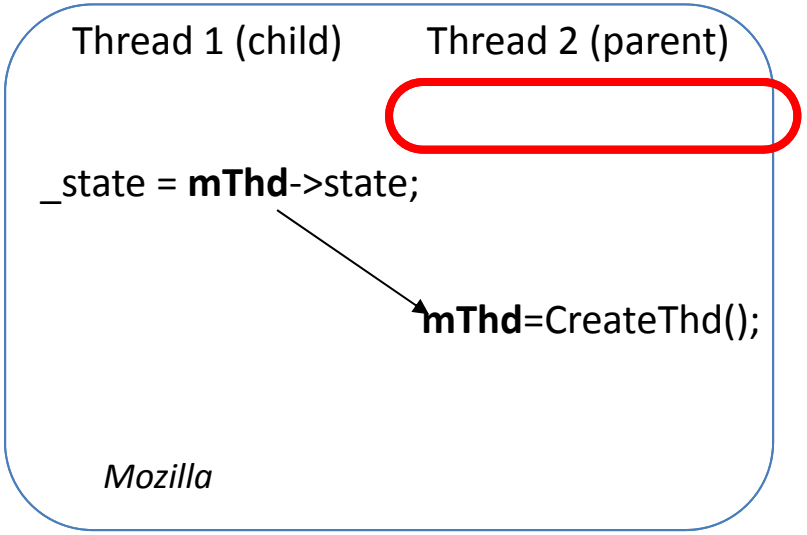
- Stop before every shared-variable write?
  - When the previous access is abnormal?
- Stop before every shared-variable read?
  - When the previous access is abnormal?

Our solution – A(nticipating) I(nvariant)

For an instruction i,
a fixed set of instructions P are expected to
precede it and touch the same variable
from a different thread
(for correct execution)

Example


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


Thread 1 (child)
Thread 2 (parent)
_state = mThd->state;
mThd=CreateThd();
Mozilla

Example

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

MySQL

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

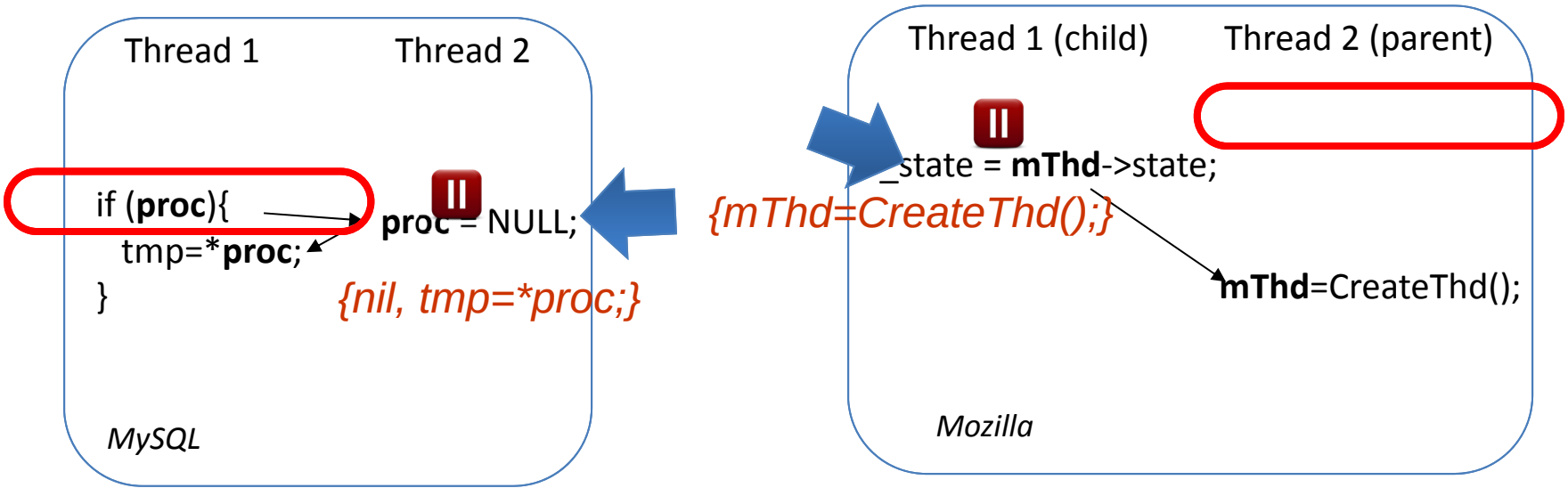
MySQL

Thread 1 (child)	Thread 2 (parent)
	mThd=CreateThd();
_state = **mThd**->state;	

Mozilla

Our solution --- the whole story

- Step 1
  - Off-line training to obtain AI invariants
- Step 2
  - On-line monitoring to look for AI violation
- Step 3
  - Stall a thread when AI invariant is violated



Evaluation of AI

- Evaluated on a large number of bugs and software
  - 35 real-world bugs from 10+ applications
- Prevent all 35 concurrency-bug failures
- Training
  - ~100 for small applications, ~1000 for large applications
- Runtime overhead
  - <5% for desktop & I/O intensive applications
  - >1000% for scientific computing applications

Conclusions

- ConAir and AI complement each other
  - Reactive vs. proactive
  - Effect-guided vs. cause-guided
  - …
- Prevention and recovery are promising!