Deterministic User-level Replay of Concurrent Programs

Joint work with Jeff Huang, Andy Zhou, Peng Liu, Richard Xiao
Prism Research Group
Computer Science and Engineering
The Hong Kong University of Science and Technology
(HKUST)

Outline

- What's replay?
- The challenges of user-level deterministic replay
- Our contributions
  - LEAP (FSE 10)
  - STRIDE (ICSE 12)
  - CLAP (PLDI SRC)
- Conclusions

What is replay?

- Scope
  - Programs where threads randomly scheduled on multiprocessors and communicate through shared memory
- Definition
  - To repeat a previously exercised execution according to a diagnostic criterion
    - Full execution → Time travelling virtual machine [King et al. ATC05]
    - Computed value → iDNA [Bhansali et al. VEE 06]
    - Failure reproduction → ODR [Altekar and Stoica SOSP 09], our approaches
- Basic mechanism
  - Record phase ➔ runtime monitoring to produce a replay log
  - Replay phase ➔ use the replay log to reconstruct the recorded execution
- Research
  - [LeBlanc et al TOCS 1986] ➔ [Zhou et al ICSE 2012]
  - Spanning many areas ➔ FSE/ICSE/OOPSLA/PLDI/OSDI/ASPLOS/MICRO/VEE

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Replay Implementation

- Implementation methods
  - Hardware based
    - Piggyback cache coherence messages [FDR’03, ReRun’09, DeLorean’09]
    - Invariant timestamp on x86 TSO [CoreRacer’11]
  - Software based
    - User-level. Program instrumentation [LEAP’10, STRIDE’12]
    - System level. VM-Level logging [SMP-ReVirt’08] [King et al. ATC 05]
- Replay guarantees
  - Deterministic → guarantee to reproduce an earlier run [LEAP’10, STRIDE’12, ODR’09]
  - Probabilistic → replay with a best effort (high probability) [PRES’09, ESD’10]

Our focus: deterministic user-level replay for bug reproduction

- Deterministic
  - Determinism helps the diagnostic process more effectively
  - Many client analyses require deterministic replay
    - Iterative debugging
    - Fault localization
    - Bug classification
- User level
  - Easy to use: Instrument → Monitor→Log → Replay
  - Require no OS/Hardware modifications
- Bug reproduction
  - Focus on producing the order of race/RW dependency
  - Relax on reproducing original schedule and computed values.

Replay Exercise

- T1:

  1 local_X= G ;

  2 local_X++;

  3 G =local_X;

- T2:

  4 local_Y= G ;

  5 local_Y++;

  6 G =local_Y;

- Initial: G=0

Replay Exercise

- T1:

G=0

1 local_X= G ;

2 local_X++;

G=1

3 G =local_X;

- T2:

4 local_Y= G ;

5 local_Y++;

G=2

6 G =local_Y;

- Initial: G=0

G is 2

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Replay Exercise

- T1:

G=0 1 local_X= G ;

2 local_X++;

3 G =local_X;

- T2:

Key to restore the race order

G=1 4 local_Y= G ;

5 local_Y++;

G=1 6 G =local_Y;

- Initial: G=0

G is 1

Solution 1: Record the order of race

- T1:

  1 local_X= G ;

  2 local_X++;

  3 G =local_X;

  Record()

  3, 4

- T2:

  4 local_Y= G ;

  Record()

  4, 3

  5 local_Y++;

  6 G =local_Y;

Solution 1: Order-based replay

- T1:

  1 local_X= G ;

  2 local_X++;

  3 G =local_X;

  Record()

- T2:

  4 local_Y= G ;

  Record()

  5 local_Y++;

  6 G =local_Y;

Solution 2: Search (infer) the order of race

- T1:

Read, 0 → 1 local_X= G ;

2 local_X++;

Write, 1 → 3 G =local_X;

- T2:

4 local_Y= G ; ← Read, 0

5 local_Y++;

6 G =local_Y; ← Write, 1

Given thread-local load/store value trace, is there a schedule that is, e.g. sequentially consistent.

An NPC problem. [Gibbons et al. J.SIAM 97]

Solution Spectrum

- Order-based
  - Explicitly tracking RW dependencies, i.e., order of race
  - Guarantees to replay the execution
  - We need to use locks, a lot of them
- Search-based
  - Infer RW dependencies through load/store values
  - Require no use of locks
  - NPC problem. Non-deterministic
- Prior art of user-level deterministic replay
  - Primarily order-based
  - Representative techniques: Dejavu, Recplay

LEAP: a local order-based approach

Thread t1

```
{
    1: x=1
    2: y=1
    3: if(x<0)
    4:   ERROR
}
```

Thread t2

```
{
    5: y=0
    6: if(y=1)
    7:   x=-1
}
```

schedule: 1->5->2->6->7->3->4->ERROR

LEAP: a local order-based approach

Thread t1

```
1: x=1
2: y=1
3: if(x<0)
4:    ERROR
```

Thread t2

```
5: y=0
6: if(y=1)
7:    x=-1
```

Per-SPE access vector:

x

y

LEAP: a local order-based approach

schedule: 1->5->2->6->7->3->4

Thread t1	Thread t2
1: x=1	5: y=0
2: y=1	6: if(y=1)
3: if(x<0)	7: x=-1
4: ERROR	

Per-SPE access vector:

x	t1	t2	t1
y	t2	t1	t2

LEAP: a local order-based approach

schedule: 1->5->2->6->7->3->4

Thread t1	Thread t2
1: x=1	5: y=0
2: y=1	6: if(y=1)
3: if(x<0)	7: x=-1
4: ERROR	

6 local syncs!

Per-SPE access vector:

x		
t1	t2	t1

y		
t2	t1	t2

LEAP: how to replay?

Enforce the same access order to each shared variable

x	t1	t2	t1
y	t2	t1	t2

Thread t1

1: x=1

2: y=1

3: if(x<0)

4:    ERROR

Thread t2

5: y=0

6: if(y=1)

7:    x=-1

LEAP: how to replay?

Enforce the same access order to each shared variable

x	t1	t2	t1
y	t2	t1	t2

Thread t1

1: x=1

2: y=1

3: if(x<0)

4:    ERROR

Is t1's turn to access x?

Yes, go ahead!

Thread t2

5: y=0

6: if(y=1)

7:    x=-1

LEAP: how to replay?

Enforce the same access order to each shared variable

x t2 t1

y t2 t1 t2

Thread t1

1: x=1

2: y=1

3: if(x<0)

4: ERROR

Is t1's turn to access y?

No, postpone!

Thread t2

5: y=0

6: if(y=1)

7: x=-1

LEAP: how to replay?

Enforce the same access order to each shared variable

x | t2 | t1 |

y | t2 | t1 | t2 |

Thread t1

1: x=1

2: y=1

3: if(x<0)

4: **ERROR**

Is t2's turn to access y?

Yes, go ahead!

Thread t2

5: y=0

6: if(y=1)

7: x=-1

LEAP: how to replay?

Enforce the same access order to each shared variable

x	t2	t1	y	t1	t2

Thread t1

1: x=1

2: y=1

3: if(x<0)

4: ERROR

Is t2's turn to access y?

No, postpone!

Thread t2

5: y=0

6: if(y=1)

7: x=-1

LEAP: how to replay?

Enforce the same access order to each shared variable

t2 exits

Thread t1	Thread t2
1: x=1	5: y=0
2: y=1	6: if(y=1)
3: if(x<0)	7: x=-1
4: ERROR	

Error triggered!

Leap Characteristics

- A state of the art order-based deterministic replay technique
  - Lightweight compared to existing approaches.
  - Heavy use of static analysis and bytecode transformation
  - Formal proof of replay correctness
  - First fully automated tool available to the public
- Weaknesses
  - Too many synchronizations
  - Low level data races disappear even the benign ones
  - Sequentially consistent execution

CLAP: a solver-based approach

Completely no sync & better schedule!

- **No sync**
- **Record branch choices only**
- **Compute schedule by solving constraints**

CLAP: a solver-based approach

1. Record thread local path at runtime
2. Construct the execution constraints using symbolic execution
3. Use a SMT solver to compute schedule

Path constraints
Program order constraints
Partial order constraints
Read-write constraints

→ **Constraint Solver** → **Global order of shared variable accesses**

Path profiling is lightweight

i.e. ~31% overhead with Ball-Larus algorithm

CLAP: a solver-based approach

schedule: 1->5->2->6->7->3->4

Thread t1	**Thread t2**
1: x=1	**5: y=0**
2: y=1	**6: if(y=1)**
3: if(x<0)	**7: x=-1**
4: ERROR	

Branches: t1 $\boxed{3\text{->}4}$ t2 $\boxed{6\text{->}7}$

CLAP: a solver-based approach

schedule: 1->5->2->6->7->3->4

Thread t1	Thread t2
1: x=1	5: y=0
2: y=1	6: if(y=1)
3: if(x<0)	7: x=-1
4: ERROR	

Branches: t1 3->4 t2 6->7

Paths: t1 1->2->3->4 t2 5->6->7

CLAP: a solver-based approach

Thread t1

1: x=1

2: y=1

3: if(x<0)

4: ERROR

Thread t2

5: y=0

6: if(y=1)

7: x=-1

CLAP: Symbolic Encoding

SV variables – *instances of shared variables*
S-variables - *symbolic value returned by remote read :*
O-variables - *order of SV variables, e.g., X1:1 => Ox1:1*

Thread t1		**Thread t2**	
1: x=1	x1:1 = 1 ?	**5: y=0**	y2:1 = 0
2: y=1	y1:1 = 1	**6: if(y==1)**	y2:2 = Sy2:2
3: if(x<0)	x1:2 = Sx1:2	**7: x=-1**	x2:1 = -1 ?
4: ERROR			

Path constraints: (Sx1:2 < 0) & (Sy2:2 == 1)

Order constraints: Ox1:1<Oy1:1<Ox1:2 Oy2:1<Oy2:2<Ox2:1

Read-write constraints: Sx1:2 = x1:1 & (Ox2:1<Ox1:1 | Ox2:1>Ox1:2)
Sx1:2 = x2:1 &Ox1:1<Ox2:1<Ox1:2

Path constraints	Sx1:2<0	Sy2:2 ==1
Order constraints	Ox1:1<Oy1:1<Ox1:2	Oy2:1<Oy2:2<Ox2:1

Read-write constraints

Sx1:2 = x1:1 & (Ox2:1<Ox1:1 | Ox2:1>Ox1:2) ||
x2:1 & Ox1:1<Ox2:1<Ox1:2

Sy2:2 = y2:1 & (Oy1:1<Oy2:1 | Oy1:1>Oy2:2) ||
y1:1 & Oy2:1<Oy1:1<Oy2:2

↓

solver

Path constraints Sx1:2<0 Sy2:2 ==1

Order constraints Ox1:1<Oy1:1<Ox1:2 Oy2:1<Oy2:2<Ox2:1

Read-write constraints

Sx1:2 = x1:1 & (Ox2:1<Ox1:1 | Ox2:1>Ox1:2) ||
x2:1 & Ox1:1<Ox2:1<Ox1:2

Sy2:2 = y2:1 & (Oy1:1<Oy2:1 | Oy1:1>Oy2:2) ||
y1:1 & Oy2:1<Oy1:1<Oy2:2

↓

solver

↓

Ox1:1 = 2

Oy1:1 = 3

Ox1:2 = 6

Oy2:1 = 1

Oy2:2 = 4

Ox2:1= 5

Path constraints	Sx1:2<0	Sy2:2 ==1
Order constraints	Ox1:1<Oy1:1<Ox1:2	Oy2:1<Oy2:2<Ox2:1

Read-write constraints

Sx1:2 = x1:1 & (Ox2:1<Ox1:1 \| Ox2:1>Ox1:2) \|\|
x2:1 & Ox1:1<Ox2:1<Ox1:2
Sy2:2 = y2:1 & (Oy1:1<Oy2:1 \| Oy1:1>Oy2:2) \|\|
y1:1 & Oy2:1<Oy1:1<Oy2:2

solver

Thread t1

1: x=1 Ox1:1 = 2

2: y=1 Oy1:1 = 3

3: if(x<0) Ox1:2 = 6

Thread t2

5: y=0 Oy2:1 = 1

6: if(y=1) Oy2:2 = 4

7: x=-1 Ox2:1= 5

Path constraints	Sx1:2<0	Sy2:2 ==1
Order constraints	Ox1:1<Oy1:1<Ox1:2	Oy2:1<Oy2:2<Ox2:1

Read-write constraints

Sx1:2 = x1:1 & (Ox2:1<Ox1:1 | Ox2:1>Ox1:2) ||
x2:1 & Ox1:1<Ox2:1<Ox1:2
Sy2:2 = y2:1 & (Oy1:1<Oy2:1 | Oy1:1>Oy2:2) ||
y1:1 & Oy2:1<Oy1:1<Oy2:2

solver

Thread t1		**Thread t2**	
1: x=1	Ox1:1 = 2	**5: y=0**	Oy2:1 = 1
2: y=1	Oy1:1 = 3	**6: if(y==1)**	Oy2:2 = 4
3: if(x<0)	Ox1:2 = 6	**7: x=-1**	Ox2:1= 5

schedule: 5->1->2->6->7->3->ERROR

CLAP Characteristics

- **Reduce multiprocessor replay to solving two well known problems**
  - Thread-local profiling
  - Automatic theorem proving
- **The schedule computed by CLAP is guaranteed to reproduce the bug**
- **Applicable to TSO/PSO models**
  - Relax thread-order constraints
- **Encode the context switch bound to reduce the search space**
  - Sum of the delta of thread-local consecutive order variables < Context switch bound

CLAP: recording overhead

Program	Native	Time (Overhead) LEAP	CLAP	▽	Log LEAP	CLAP	▽
sim_race	2ms	4ms(-)	4ms(-)	-	448B	126B	↓72%
bbuf	2ms	6ms(-)	4ms(-)	↓33%	12.2K	1.1K	↓91%
swarm	68ms	0.770s(1032%)	0.101s(48.5%)	↓87%	9.20M	215.6K	↓97.7%
pbzip2	0.140s	0.170ms(21.4%)	0.153ms(9.3%)	↓10%	19.5K	1.8K	↓91%
aget	0.231s	0.490s(112%)	0.270s(17%)	↓45%	683.8K	24.3K	↓96.4%
pfscan	0.135s	1.537s(1172%)	0.260s(92.6%)	↓83.1%	1.61M	330.5K	↓79.5%
racey	0.262s	11.5s(4289%)	0.705(269%)	↓93.9%	68.2M	3.81M	↓94.4%

- Runtime overhead: 9.3%-269%
- Time: 10%-93.9% reduction of LEAP
- Space: 72%-97.7% reduction of LEAP

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Conclusion

- Challenges of user-level deterministic replay
  - Recording overhead
  - Heisenberg effect
  - Replay complexity
- Two techniques
  - LEAP, a state of the art order-based technique
  - CLAP, light weight recording through SMT solvers
- Future work
  - Event driven systems.
  - Long running systems
  - Distributed systems.

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Thank you very much !

More information:
http://www.cse.ust.hk/~charlesz