Applying Hardware Transactional Memory for Concurrency-Bug Failure Recovery in Production Runs

Yuxi Chen, Shu Wang, Shan Lu, and Karthikeyan Sankaralingam*
What are the concurrency bugs?

- Synchronization mistakes in multithreaded programs

Thread 1

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

Thread 2

```c
ptr = NULL;
```

Segfault
Concurrency bugs are problematic

- Common
- Hard to diagnose and fix correctly
- Disasters in production runs
Failure recovery in production run

- Semantic correctness
- Performance
  - Low recovery latency
  - Low overhead

So what’s the solution?
Rollback and reexecution can help recovery

How to rollback and reexecute correctly with low overhead?
Past solution I

Traditional Roll-back Recovery

Thread 1

Thread 2

Thread 3

Execution

Whole-program memory-state checkpoint

Failure site
Rx: Treating bugs as allergies – a safe method to survive software failure, SOSP’05
Past solution II

ConAir Roll-back Recovery

Thread 1

Thread 2

Thread 3

setjmp

longjmp

Execution

Failure site
ConAir

- Poor capability
- Good performance

Recovery capability

Overhead (%)

0.5

10

Rx

- Good capability
- Poor performance

ConAir

- Poor capability
- Good performance
ConAir’s recovery capability limitation

- ConAir cannot reexecute shared-variable writes

```cpp
if(ptr){
    *buf = newBuf();
    tmp = *ptr;
}
ptr = NULL;
```

Heap

- *buf_old = 0x123
- *buf_new = 0x234

Segment fault (Segfault)
How can we do better?

- **ConAir**: worse capability, better performance
- **Rx**: better capability, worse performance

Diagram:
- Y-axis: Recovery capability
- X-axis: Overhead (%)
Our contribution: \textbf{BugTM}_H (Hardware)

Leveraging HTM

Recovery capability vs. Overhead (%)

- ConAir
- \textbf{BugTM}_H
- Rx

- Xbegin
- Xabort
- Xend
Our contribution: BugTM_{HS} (Hardware-Software Hybrid)

BugTM static analysis
code transformation

FREE LUNCH!
Outline:
1. $\text{BugTM}_H$
2. $\text{BugTM}_{HS}$
3. Evaluation Methodology
4. Experiment Results
5. Conclusion
HTM

- Implicit checkpoint
- Rollback-reexecution

OPPORTUNITY

CHALLENGE
Challenges

- Performance challenges
  - High frequency of transaction uses
  - Unsuitable content of transactions (e.g., trapping instructions)
  - Nesting && Loops
Challenges

- Performance challenges
- Correctness challenges
  - Unpaired transaction-start and transaction-commit
  - Deterministic aborts (e.g. trapping instruction aborts)
Challenges

- Performance challenges
- Correctness challenges
- Failure recovery challenges
  - Surround the buggy codes when failures happen
  - HTM-abort handlers

1. Carefully design HTM start, commit, and abort routines
2. Selectively insert HTM start, commit, and abort routines
Intel hardware transaction memory -- TSX

- Xbegin
- Xend
- Xtest
- Xabort

Thread 1

+ Xbegin();
+ Xtest();
+ assertion();
+ Xabort();
+ Xend();
Routines Design

```c
mXbegin() {
    if (!Xtest())
        Xbegin();
}

mXend() {
    if (Xtest())
        Xend();
}
```

No nested TM!

No unpaired Xbegin and Xend during run time!
Where to put mXabort

Principle: put mXabort before where failures might happen

```
If(ptr){
  *buf = newBuf();
  if(ptr == NULL){
    +mXabort();
    *buf__asswBfffl();
  }
  else{ p = *ptr;
    tmp = *ptr;
  }
}
```
Where to put mXbegin

Principle:
- Avoid trapping instructions abort
- Minimize capacity abort

```c
time();
+ mXbegin();
if(ptr){
    *buf = newBuf();
    if(Ptr==NULL){
        mXabort();
        __assert_fail;
    }else{
        tmp = *ptr;
    }
}
```
Where to put mXbegin

- CFG node
- Potential failure
- Trapping/Call/loop-exit
  Instruction or function entrance
Where to put mXend

Principles:

- Avoid trapping instructions abort
- Minimize capacity abort
  - End TM before loop entry
  - End TM before function exit

```c
time ();
+ mXbegin();
If(ptr){
    *buf = newBuf();
    if(Ptr==NULL){
        + mXabort();
        __assert_fail;
    }else{
        tmp = *ptr;
    }
}
+ mXend();
```
Where to put mXend

- mXend() node
- CFG node
- Potential failure
- Trapping/Call/loop-header instruction
- Function exit
Design fallback and retry

Principle: Reexecution only when aborts might be caused by concurrency bugs

- Concurrency bug relevant aborts (reexecute in Tx mode)
  - Data conflict abort && Xabort abort

- Concurrency bug irrelevant aborts (reexecute in non-Tx mode)
  - Capacity abort && Trapping instruction abort
Outline:
1. BugTM\textsubscript{H}
2. BugTM\textsubscript{HS}
3. Evaluation Methodology
4. Experiment Results
5. Conclusion
• **Where to put setjmp and mXbegin**

\[ \text{Loc}_{\text{setjmp}} : \text{locations where ConAir inserts setjmp} \]
\[ \text{Loc}_{\text{mXbegin}} : \text{locations where BugTM}_H \text{ inserts mXbegin} \]

- Insert setjmp at every \( \text{Loc}_{\text{setjmp}} \)
- Insert mXbegin **only** when \( \text{Loc}_{\text{mXbegin}} \) is farther than \( \text{Loc}_{\text{setjmp}} \)
- **Not** insert mXbegin if \( \text{Loc}_{\text{setjmp}} \) and \( \text{Loc}_{\text{mXbegin}} \) are same
- **Where to put mXend**
  - Exactly same as BugTM$_H$
• **How to retry**

- HTM rollback first (under an active transaction)
  - longer reexecution region

- Longjmp rollback (not under an active transaction)
  - If HTM rollback fails, longjmp rollback can still have a chance
Outline:
1. $\text{BugTM}_H$
2. $\text{BugTM}_{HS}$
3. Evaluation Methodology
4. Experiment Results
5. Conclusion
Evaluation methodology

- Benchmarks (29 bugs$^{[1,2,3,4,5,6]}$)

- micro architecture
  Broadwell (4-core Intel Core i7-5775C)
- LLVM

## Recovery capability

### Recovery capability comparison

<table>
<thead>
<tr>
<th>Benchmarks(ID)</th>
<th>Root Cause</th>
<th>ConAir</th>
<th>BugTM&lt;sub&gt;H&lt;/sub&gt;</th>
<th>BugTM&lt;sub&gt;HS&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mysql2011</td>
<td>RAR Atomicity Violation</td>
<td>-</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Mysql38883</td>
<td>RAR Atomicity Violation</td>
<td>-</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Apache21287</td>
<td>RAW Atomicity Violation</td>
<td>-</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Moz-JS18025</td>
<td>RAW Atomicity Violation</td>
<td>-</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Moz-JS142651</td>
<td>RAW Atomicity Violation</td>
<td>-</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Bank</td>
<td>WAR Atomicity Violation</td>
<td>-</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Transmission</td>
<td>Order Violation</td>
<td>✔</td>
<td>-</td>
<td>✔</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td><strong>1</strong></td>
<td><strong>6</strong></td>
</tr>
</tbody>
</table>

ConAir < BugTM<sub>H</sub> < BugTM<sub>HS</sub>
Performance Overhead

ConAir > BugTM_{HS} > BugTM_{H}

overhead comparing with baseline: %
Outline:
1. $\text{BugTM}_H$
2. $\text{BugTM}_{HS}$
3. Evaluation Methodology
4. Experiment Results
5. Conclusion
Conclusions

- BugTM can help recover all major types of concurrency-bug failures in production run
  - Low run-time overhead
  - Outperform the state of art approach (ConAir)
  - Present novel ways of using HTM techniques (failure recovery)

Come and eat this free lunch!
Thank you!
Q & A