Detecting and Avoiding Concurrency Bugs

Pil Jae Jang
Cyril Agbi
Paper similarities

- Testing Parallel programming is hard to debug due to interleaving bugs

- A viable solution is to better equip programmers to detect and fix these bugs

- Praise Transactional Memory
Learning from Mistakes

— A Comprehensive Study on Real World Concurrency Bug Characteristics
Motivation

Writing correct concurrent programs is difficult! why?

a. Concurrency bug detection - imperfect
   i. Most of research: single variable, changing lock

b. Concurrent program testing and model checking
   i. Exponential interleaving

c. Concurrent programming language design
   i. TM provides programmers an easier way to specify which code regions should be atomic. But, Not perfect!
Notes

Deals with Four applications
- Not cover all applications

Data Race
- All of Data Races are not bug.
- Ex) benign race such as “while-flag”
Bug pattern study

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>Atomicity</th>
<th>Order</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>14</td>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apache</td>
<td>13</td>
<td>7</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>41</td>
<td>29</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td><strong>74</strong></td>
<td><strong>51</strong></td>
<td><strong>24</strong></td>
<td><strong>2</strong></td>
</tr>
</tbody>
</table>
Atomicity

Thread 1
S1: if (thd->proc_info)
{
    S2: fputs(thd->proc_info, ...);
}

Thread 2
... S3: thd->proc_info=NULL;
...

MySQL ha_innodb.cc

Buggy Interleaving
Other - rare

Thread 1

```c
void buf_flush_try_page() {
    ...
    rw_lock(&lock);
}
```

Thread 2 … Thread n

```c
rw_lock(&lock);
```

Monitor thread

```c
void error_monitor_thread() {
    if(lock_wait_time[i] > fatal_timeout)
        assert(0, "We crash the server; It seems to be hung.");
}
```

MySQL buf0flu.c
MySQL srv0srv.c

- Originally for deadlock detection
- Ideally need to set `fatal_timeout=infinite` to detect deadlock
Order

Thread 1

void init (...)
{
...

mThread = PR_CreateThread (mMain, ...);
...
}

Thread 2

void mMain (...)
{

mState = mThread->State;
...
}

Correct Order

Buggy Order

Thread 2 should not dereference mThread before Thread 1 initializes it.

Mozilla nsthread.cpp

- Between Write and Read
Order

- Between Write and Write
- can hang forever
Order

**Thread 1**
void js_DestroyContext (…) {
    /* last one entering this function */
    js_UnpinPinnedAtom(&atoms);
}

**Thread 2**
void js_DestroyContext (…) {
    /* non-last one entering this function */
    js_MarkAtom(&atoms, …);
}

- Between two groups

Mozilla jsctxt.c, jsgc.c

Correct Order
js_UnpinPinnedAtom should happen after js_MarkAtom.
Otherwise, program crashes.

Buggy Order
Bug manifestation study

How many threads are involved?
Most (101 out of 105) concurrency bugs involves only two threads.

• Increase the workload, then check pairs of threads.
• Few concurrency bugs would be missed.
Bug manifestation study

How many variables are involved?
66% -> One variable
34% -> More than one variable
One variable

Thread 1
S1: if (thd->proc_info)
{
    S2: fputs(thd->proc_info, ...);
}

Thread 2
S3: thd->proc_info=NULL;
...

MySQL ha_innodb.cc

Buggy Interleaving
One variable

Thread 1

```cpp
void init (...) {
    ... 
    mThread = PR_CreateThread (mMain, ...);
    ... 
}
```

Thread 2

```cpp
void mMain (...) {
    mState = mThread->State;
    ... 
}
```

Correct Order

Wrong Order

Thread 2 should not dereference mThread before Thread 1 initializes it.

Mozilla nstread.cpp
One variable

**Thread 1**
```c
int ReadWriteProc (⋯)
{
    ...
    S1: PBReadAsync ( &p);
    S2: `io_pending` = TRUE;
    ...
    S3: while (`io_pending`) {⋯};
    ...
}
Mozilla macio.c
```

**Thread 2**
```c
void DoneWaiting (⋯)
{
    /*callback function of PBReadAsync*/
    ...
    S4: `io_pending` = FALSE;
    ...
}
Mozilla macthr.c
```

**Correct Order**

**Buggy Order**

S4 is assumed to be after S2. If S4 executes before S2, thread 1 will hang.
More than one variable

**Thread 1**

```c
void nsTextFrame::PaintAsciiText(...) {
    :
   putc(mContent[mOffset+mLength-1]);
    :
}
```

`nsTextFrame.cpp`

**Thread 2**

```c
void nsPlaintextEditor::Cut() {
    /* change the mContent */
    nsPlaintextEditor.cpp
    void nsTextFrame::Reflow("
    {
        /* calculate and then set correct mOffset and mLength */
    }
```

`nsMsgSend.cpp`

**Buggy Interleaving**

mContent, mOffset and mLength are inconsistent in the middle of Cut and Reflow.

Paint at this moment might lead to crash.

---

`mContent, mOffset, mLength` are shared

mOffset and mLength together mark a “valid region” inside mContent string.
More than one variable

- The required condition for the bug manifestation is that thread 1 uses the three correlated variables in the middle of thread 2’s modification to these three variables.

- We need new concurrency bug detection tools to address multiple variable concurrency bugs.

- Most existing bug detection tools only focus on single-variable concurrency bugs.
How many accesses are involved?

<table>
<thead>
<tr>
<th>Non-deadlock concurrency bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
</tr>
<tr>
<td>MySQL</td>
</tr>
<tr>
<td>Apache</td>
</tr>
<tr>
<td>Mozilla</td>
</tr>
<tr>
<td>OpenOffice</td>
</tr>
<tr>
<td>Overall</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deadlock concurrency bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
</tr>
<tr>
<td>MySQL</td>
</tr>
<tr>
<td>Apache</td>
</tr>
<tr>
<td>Mozilla</td>
</tr>
<tr>
<td>OpenOffice</td>
</tr>
<tr>
<td>Overall</td>
</tr>
</tbody>
</table>
How many accesses are involved?

• Significant implication for concurrent program testing.
  – The challenge in concurrent program testing is that the number of all possible interleavings is \textit{exponential to the number of dynamic memory accesses}, which is too big to thoroughly explore.

• Exploring all possible orders \textit{within every small groups} of memory accesses, e.g. groups of 4 memory accesses.
  – The complexity of this design is only polynomial to the number of dynamic memory accesses, which is a huge reduction from the exponential-sized all-interleaving testing scheme.
Bug manifestation study
Take away

\[
\begin{array}{cccccc}
\text{thread 0} & \text{thread 1} & \text{thread 2} & \ldots & \text{thread 99} \\
\end{array}
\]

100 \( C_2 \) pairs

vs

\[100 C_2 + 100 C_3 + \ldots + 100 C_{100} = 2^{100} - 3\]
Bug fix study

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>COND</th>
<th>Switch</th>
<th>Design</th>
<th>Lock</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Apache</td>
<td>13</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>41</td>
<td>13</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Overall</td>
<td>74</td>
<td>19</td>
<td>10</td>
<td>19</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

- Adding Lock cannot enforce order intention.
Bug fix study

(1) Condition check (denoted as COND):
Ex) use while-flag to fix order-related bugs consistency

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Correct Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>void js_DestroyContext (...) {</td>
<td>void js_DestroyContext (...) {</td>
<td>Buggy Order</td>
</tr>
<tr>
<td>/* last one entering this function */</td>
<td>/* non-last one entering this function */</td>
<td>js_UnpinPinnedAtom should happen after js_MarkAtom.</td>
</tr>
<tr>
<td>js_UnpinPinnedAtom(&amp;atoms);</td>
<td>js_MarkAtom(&amp;atoms, ...);</td>
<td>Otherwise, program crashes.</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>Mozilla jsctxtx.c, jsgc.c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bug fix study

(1) Condition check (denoted as COND):

Ex) if(strlen(mContent) >= mOffset + mLength)
Bug fix study

(2) Code switch (denoted as Switch)

```
Thread 1
int ReadWriteProc (…) {
    …
    S1: PBReadAsync ( &p);
    S2: io_pending = TRUE;
    …
    S3: while ( io_pending ) {…};
    …
}

Mozilla macio.c

Thread 2
void DoneWaiting (…) {
    /*callback function of PBReadAsync*/
    …
}

Mozilla maclhr.c
```

Correct Order

Buggy Order

S4 is assumed to be after S2. If S4 executes before S2, thread 1 will hang.

(3) Algorithm/Data-structure design change

ex) remove some variable from class that does not need to be shared.
## Discussion: bug avoidance

**Transactional memory (TM)**

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>Can Help</th>
<th>TM might help</th>
<th>Little Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>23</td>
<td>7</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Apache</td>
<td>17</td>
<td>7</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Mozilla</td>
<td>57</td>
<td>25</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>105</td>
<td>41</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

**Table 10.** Can TM help avoid concurrency bugs?
Discussion: bug avoidance

Transactional memory (TM)

- Atomicity violation bugs and deadlock bugs with relatively small and simple critical code regions can benefit the most from TM, which can help programmers clearly specify this type of atomicity intention.

- Figure 8 shows an example, where programmers use a consistency check with re-execution to fix the bug. Here, a transaction(with abort, rollback and replay) is exactly what programmers want.
Discussion: bug avoidance

Concern with Transactional Memory

• I/O operations:
As operations like I/O are hard to roll back, it is hard to use TM to protect the atomicity of code regions which include such operations.

• Too large memory footprint:
Mozilla bugs include the whole garbage collection process. These regions could have too large memory footprint to be effectively handled by hardware-TM
Discussion: bug avoidance

Problem with Transactional Memory

• The basic TM designs cannot help enforce the intention that “A has to be executed before B”. Therefore, they cannot help avoid many related order-violation bugs
Conclusions and future work

• Design new bug detection tools to address multiple-variable bugs and order violation bugs.

• can pairwisely test concurrent program threads and focus on partial orders of small groups of memory accesses to make the best use of testing effort.

• can have better language features to support “order” semantics to further ease concurrent programming.
A Case for an Interleaving Constrained Shared-Memory Multi-Processor
Motivation

Writing parallel programs is hard because….

• INTERLEAVING
  – Verifying simple contracts is NP-complete
  – Hard to guarantee correctness
  – Hard to debug

Proposed Solution…

• Predecessor Set (PSet)
  – Constrain program to follow tested interleavings (that are good)
  – Better runtime consistency and easier to debug
Motivation - PSet

Tools that are capable of detecting:

- Data Races
  - Happens-before based vs lockset based detectors
  - Benign data races
- Atomicity Violations
  - Most tools rely on programmer to specify atomic regions
- Ordering violations

Current tools not good at detecting all three, but...

**PSet is capable of detecting ALL THREE**
How PSet Works

• For each RW section in a thread, a PSet contains the set of all dependencies from other threads that can occur before it
• On each RW section, checks to see if the last RW to memory location is in current section’s PSet. If not…

1. **STALL**: The thread will stall until one of the section’s predecessors completes.

2. **CHECKPOINT & ROLLBACK**: The program returns to a checkpoint and re-executes.
Implementation

- **Instruction**
  - **P**
  - **Size**
  - **PInst1**
  - **PInst2**
  - **PInst3**

- **PSet Type (2 bits)**
  - 00 - Not Tested
  - 01 - Null Pset
  - 10 - PSet Size = 1
  - 11 - PSet Size > 1

- **PSet Size (1 byte)**

- **Predecessor Address (4 bytes)**
  - **Lib ID**
  - **Rel Addr**
    - 4 bits
    - 28 bits
Notes on PSet

PSets have a worse case space complexity of $O(N^2)$
  • But about 95% of instructions have no PSet

Implementing reset requires a lot of additional architecture
  • Add pset instructions to ISA
  • Space to track last reader/writer as well as PSet constraints

The constraints need to be acquired through learning before runtime
Notes on PSet

Violation handling isn’t full-proof:

- Stalling can enter a deadlock scenario
  - Solution: Time-out scheme (thread resumes after timeout)
- There is no good tested interleave path at checkpoint
  - Solution: After some number of tries, go back to further checkpoint

Design specified in paper “does not account for the interleavings between two or more memory operations accessing different memory locations.”
## Results

<table>
<thead>
<tr>
<th>Bug #</th>
<th>Program</th>
<th>Type</th>
<th>Stall</th>
<th>Rollback</th>
<th>True Constraint Violations</th>
<th>False Constraint Violations</th>
<th>Rollback Window Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Static</td>
<td>Dynamic</td>
<td>Static</td>
</tr>
<tr>
<td>1</td>
<td>Pbzlip2</td>
<td>Real</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Aget</td>
<td>Real</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Pfscan</td>
<td>Injected</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Apache</td>
<td>Real</td>
<td>No</td>
<td>Yes</td>
<td>2</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>MySQL</td>
<td>Real</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>MySQL</td>
<td>Extract</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Mozilla</td>
<td>Extract</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>Mozilla</td>
<td>Extract</td>
<td>No</td>
<td>Yes</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Mozilla</td>
<td>Extract</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>Mozilla</td>
<td>Extract</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>Mozilla</td>
<td>Extract</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>Mozilla</td>
<td>Extract</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>Mozilla</td>
<td>Extract</td>
<td>No</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>Mozilla</td>
<td>Extract</td>
<td>Yes</td>
<td>Yes</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>Mozilla</td>
<td>Extract</td>
<td>No</td>
<td>Yes</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Avoiding bugs using PSet constraints. True constraint violations are related to the bug.
# Results

<table>
<thead>
<tr>
<th>Programs</th>
<th>Stall</th>
<th>Rollback</th>
<th>Cannot Resolve</th>
<th>Total PSet Constraint Violations</th>
<th>Inst. Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbzip2</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>1.3E+9</td>
</tr>
<tr>
<td>aget</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.1E+7</td>
</tr>
<tr>
<td>pfscan</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>7.4E+7</td>
</tr>
<tr>
<td>apache</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>2.8E+8</td>
</tr>
<tr>
<td>mysql</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>9.7E+8</td>
</tr>
<tr>
<td>fft</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2.3E+8</td>
</tr>
<tr>
<td>fmm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1.6E+9</td>
</tr>
<tr>
<td>lu</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.6E+8</td>
</tr>
<tr>
<td>radix</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6.4E+7</td>
</tr>
<tr>
<td>blackscholes</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8.1E+8</td>
</tr>
<tr>
<td>canneal</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7.0E+9</td>
</tr>
</tbody>
</table>

Table 3: PSet constraint violations in bug-free executions.
Results

Figure 8: Number of test runs required for learning PSets and AVIO invariants.
Conclusion

This is only a first step!

- Capable of detecting more concurrency bugs than most other tools
  - Accomplishes the goal of allowing programmers to more reliably catch and fix concurrency bugs
- With sufficient testing, PSets can prevent concurrency bugs