SUPPORT FOR DETERMINISM IN A CONCURRENT PROGRAMMING ENVIRONMENT

Vikram Murali
“Learning from Mistakes – A Comprehensive study on Real World Concurrency Bug Characteristics”

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou, 2008
WHY THIS PAPER?

• Progress towards multicore architectures → importance and pervasiveness of concurrent programming.

• Difficulty in writing correct concurrent programs --- sequential rules don’t work here.

• Notorious Non-determinism associated with them!

• From high-end servers to desktop machines.
ADDRESSING THESE ISSUES WOULD MEAN:

EFFICIENT:

• Concurrency Bug Detection. Questionable?

• Concurrent program testing and model testing. Exponential Interleaving Space. Representative interleavings? – Con Test.

  Good understanding of manifestation critical..

• Concurrent Programming Language design.

--- THE PAPER’S GOAL.
SOME TERMINOLOGIES.

- **Data race**: Occurs when two conflicting accesses to one shared variable are executed without proper synchronization, e.g., not protected by a common lock.

- **Deadlock**: Occurs when two or more operations circularly wait for each other to release the acquired resource (e.g., locks). “Dining Philosophers!”

- **Atomicity Violation bugs**: Bugs which are caused by concurrent execution unexpectedly violating the atomicity of a certain code region.

- **Order Violation bugs**: Bugs that don’t follow the programmer’s intended order. Several undesirable effects.
METHODOLOGY

How are the bugs selected?

- Four Representative Open Source Applications: MySQL, Apache, Mozilla, OpenOffice.

- Random selection of concurrency bugs from their databases. (from over 500000 bug reports!)

- Reports with clear root cause, source code and bug fix description.

- Finally screen and choose: 105 concurrency bugs → 74 non-deadlock bugs, 31 deadlock bugs.
<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th># of Bug Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Non-Deadlock</td>
</tr>
<tr>
<td>MySQL</td>
<td>Database Server</td>
<td>14</td>
</tr>
<tr>
<td>Apache</td>
<td>Web Server</td>
<td>13</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Browser Suite</td>
<td>41</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>Office Suite</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>74</strong></td>
</tr>
</tbody>
</table>

Chosen Application set and Bug set
Bug Characteristics study divided into:

- **Bug Pattern study** → On the basis of “root causes”

- **Bug Manifestation study** → Conditions necessary and sufficient to cause a bug.
  
  -----  Conditions throw light on: threads, variables, accesses involved.

- **Bug Fix study** → Type of fix strategy employed.

VALIDITY WARNING: BEWARE OF GENERALISING!
### Bug Pattern

Patterns of non-deadlock concurrency bugs.

<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>
Finding (1): Most (72 out of 74) of the examined non-deadlock concurrency bugs are covered by two simple patterns: atomicity-violation and order-violation.

Implications: Concurrent program bug detection, testing and language design should first focus on these two major bug patterns.
Atomicity violation bug from My SQL

An order violation bug from Mozilla
Performance related: classified as neither atomicity or order violation
Finding (2): A significant number (24 out of 74) of the examined non-deadlock concurrency bugs are order bugs, which are not addressed by previous bug detection work.

Implications: New bug detection techniques are desired to address order bugs.
More Order Violation.

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*/
    ...
    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.

A write-write order violation bug from Mozilla.
Contd…

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>void js_DestroyContext (...) { /* last one entering this function */ js_UnpinPinnedAtom(&amp;atoms); }</td>
<td>void js_DestroyContext (...) { /* non-last one entering this function */ js_MarkAtom(&amp;atoms,...); }</td>
</tr>
</tbody>
</table>

A Mozilla bug that violates the intended order between two groups of operations.

• Correct Order
• Buggy Order

js_UnpinPinnedAtom should happen after js_MarkAtom.
Otherwise, program crashes.

Conclusion: Put a lock, make atomic. But no order guarantee!
BUG MANIFESTATION

• No of threads?

Finding (3): The manifestation of most (101 out of 105) examined concurrency bugs involves no more than two threads.

Implications: Concurrent program testing can pairwise test program threads, which reduces testing complexity without losing bug exposing capability much.

MAIN REASON: CONFINED PATTERN OF INTERACTION
• One Thread!

**Finding (4):** The manifestation of some (7 out of 31) deadlock concurrency bugs involves only one thread.

**Implications:** This type of bug is relatively easy to detect and avoid. Bug detection and programming language techniques can try to eliminate these simple bugs first.
The number of threads or environments involved in concurrency bugs.

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>Env.</th>
<th>&gt;2 threads</th>
<th>2 threads</th>
<th>1 thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Apache</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>41</td>
<td>1</td>
<td>0</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td><strong>74</strong></td>
<td><strong>2</strong></td>
<td><strong>1</strong></td>
<td><strong>71</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>Env.</th>
<th>&gt;2 threads</th>
<th>2 threads</th>
<th>1 thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Apache</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td><strong>31</strong></td>
<td><strong>0</strong></td>
<td><strong>1</strong></td>
<td><strong>23</strong></td>
<td><strong>7</strong></td>
</tr>
</tbody>
</table>
• Variables Involved?

**Finding (5):** 66% (49 out of 74) of the examined non-deadlock concurrency bugs involve only one variable.

**Implications:** Focusing on concurrent accesses to one variable is a good simplification for concurrency bug detection.

**REASON:** FLIP THE ORDER OF TWO ACCESSES TO DIFFERENT MEMORY LOCATIONS. DOESN'T THE PROGRAM STATE REMAIN INDEPENDENT?
• But remaining 34%?

**Finding (6):** A non-negligible number (34%) of non-deadlock concurrency bugs involve more than one variable.

**Implications:** We need new concurrency bug detection tools to address multiple variable concurrency bugs.

**Reason:** Variables can be correlated. Asynchronous access to them creates multiple variable dependency.
Mozilla – Multiple variable concurrency bug.
• Deadlock Bugs?

**Finding (7):** 97% (30 out of 31) of the examined deadlock concurrency bugs involve at most two resources.

**Implications:** Deadlock-oriented concurrent program testing can pairwise test the order among acquisition and release of two resources.
• Accesses involved?

**Finding (8.1):** 90% (67 out of 74) of the examined non-deadlock bugs can deterministically manifest, if certain orders among at most four memory accesses are enforced.

**Finding (8.2):** 97% (30 out of 31) of the examined deadlock bugs can deterministically manifest, if certain orders among at most four resource acquisition/release operations are enforced.

**Implications:** Concurrent program testing can focus on the partial order among every small groups of accesses. This simplifies the interleaving testing space from exponential to polynomial regarding to the total number of accesses, with little loss of bug exposing capability.

**REASON 8.1:** MOST OF THE EXAMINED CONCURRENCY BUGS HAVE SIMPLE PATTERNS, INVOLVE SMALL NO OF VARIABLES. EXCEPTIONS?

**REASON 8.2:** MOST OF THE EXAMINED DEADLOCK BUGS INVOLVE ONLY 2 RESOURCES.
The number of accesses or resource acquisition/release involved in concurrency bugs

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>1 acc.*</th>
<th>2 acc.</th>
<th>3 acc.</th>
<th>4 acc.</th>
<th>&gt;4 acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Apache</td>
<td>13</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>41</td>
<td>0</td>
<td>12</td>
<td>18</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td><strong>74</strong></td>
<td><strong>0</strong></td>
<td><strong>22</strong></td>
<td><strong>33</strong></td>
<td><strong>12</strong></td>
<td><strong>7</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>1 acc.*</th>
<th>2 acc.</th>
<th>3 acc.</th>
<th>4 acc.</th>
<th>&gt;4 acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Apache</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>16</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td><strong>31</strong></td>
<td><strong>7</strong></td>
<td><strong>3</strong></td>
<td><strong>20</strong></td>
<td><strong>0</strong></td>
<td><strong>1</strong></td>
</tr>
</tbody>
</table>
## 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><strong>Overall</strong></td>
<td><strong>74</strong></td>
<td><strong>19</strong></td>
<td><strong>10</strong></td>
<td><strong>19</strong></td>
<td><strong>20</strong></td>
<td><strong>6</strong></td>
</tr>
</tbody>
</table>

Fix strategies for non-deadlock concurrency bugs
REASON 1: LOCKS DON'T GUARANTEE SOME SYNCHRONISATION INTENTIONS.

REASON 2: NOT THE BEST STRATEGY, MAY INTRODUCE DEADLOCK BUGS.
Example:

Thread 1
```cpp
void NodeState::setDynamicId (int id)
{
    dynamicId = id;
}
```

Thread 2
```cpp
MgmtSrvr::status (... int *myid ...)
{
    *myid = node.m_state.dynamicId;
}
```

MySQL NodeState.hpp   MySQL MgmtSrvr.cpp

Correct Order  Buggy Order

*dynamicId should not be read before it is initialized

Wrong order will lead to wrong functionality

A MySQL bug that cannot be fixed by adding/changing locks.
SO, OTHER STRATEGIES..

1) **Condition Check**: While flag, consistency check:

```
retry:
    [...]
    n = block->n;
    [...]
    if (n != block->n)
        goto retry;
    
    btr0sea.c
```

A MySQL bug fix.

```
Thread 1
void nsTextFrame::PaintAsciiText(…)
{
    // change the mContent
    putc(mContent[mOffset+mLength-1]);
    
    nsTextFrame.cpp
    mContent, mOffset, mLength are shared

    if(strlen(mContent) >= mOffset+mLength)
```

```
Thread 2
void nsPlainTextEditor::Cut()
{
    // change the mContent
    
    nsPlainTextEditor.cpp

    void nsTextFrame::Reflow(…)
    {
        // calculate and then set correct
        // mOffset and mLength
    
    nsMsgSend.cpp
```

```
2) **Code Switch**:

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

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

Mozilla macio.c           Mozilla macthr.c
```

S1 AND S2 SWITCHED TO FIX THE BUG

3) **Algorithm and Data-structures.**
<table>
<thead>
<tr>
<th>Application</th>
<th>Total</th>
<th>GiveUp</th>
<th>Split</th>
<th>AcqOrder</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>9</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Apache</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Mozilla</td>
<td>16</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td><strong>31</strong></td>
<td><strong>19</strong></td>
<td><strong>1</strong></td>
<td><strong>7</strong></td>
<td><strong>4</strong></td>
</tr>
</tbody>
</table>

- Fix strategies for deadlock bugs

**Finding (10):** The most common fix strategy (used in 19 out of 31 cases) for the examined deadlock bugs is to let one thread give up acquiring one resource, such as a lock. This strategy is simple, but it may introduce other non-deadlock bugs.

**Implication:** We need to pay attention to the correctness of some “fixed” deadlock bugs.
ISSUES IN BUG FIXING

Aim: Programmers want to make sure js MarkAtom will not be called after js UnpinPinnedAtom. (Happens in two steps!)

(a) an incomplete fix for the bug shown in Figure 5.
This fix left a small window between S1 and S2 unprotected.

(b) a final correct fix.
Now the order between js MarkAtom and js UnpinPinnedAtom is enforced.
Transactional Memory (TM)

- RECAP.
Help from TM?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>23</td>
<td>7</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>Apache</td>
<td>17</td>
<td>7</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Mozilla</td>
<td>57</td>
<td>25</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>105</td>
<td>41</td>
<td>8</td>
<td>30</td>
</tr>
</tbody>
</table>

Can TM help avoid concurrency bugs?

**Finding (11):** TM can help avoid many concurrency bugs (41 out of the 105 concurrency bugs we examined).

**Implication:** Although TM is not a panacea, it can ease programmers correctly expressing their synchronization intentions in many cases, and help avoid a big portion of concurrency bugs.
**Finding (12):** TM can potentially help avoid many concurrency bugs (44 out of the 105 concurrency bugs we examined), if some concerns can be addressed, as shown in Table 10.

**Implication:** TM design can combine system supports and other techniques to solve some of these concerns, and further ease the concurrent programming.

**Finding (13):** 20 out of the 105 concurrency bugs that we examined cannot benefit from the basic TM designs, because the violated programmer intentions, such as order intentions, cannot be guaranteed by the basic TM.

**Implications:** Apart from atomicity intentions, there is also a significant need for concurrent programming language features to help programmers express order intentions easily.
INTERESTING?

• Bugs are very difficult to repeat : (Non-determinism in concurrent execution). Sometimes impossible. Has even resulted in guessing!

• Test cases important for bug diagnosis : A test case that can solve the above problem.

• Lack of Diagnosis tools with Programmers.
Related work, Future directions.

• Little previous work in this area! Real world concurrency bugs very hard to collect and analyse.

• “E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them” IPDPS, 2003. → gives a manipulated environment (Not real world).

• Autolocker, AtomicSet → This paper provides more motivation and platform for such work, besides improved TM.
Conclusion

• Comprehensive study, characterisation and fix strategies of real world concurrency bugs.

• Many interesting findings and implications: lot of which pivotal directions for future research.

• Creates scope for better detection, testing and concurrent programming language design.
DMP : Deterministic Shared Memory Multiprocessing

Joseph Devietti, Brandon Lucia, Luis Ceze, Mark Oskin, 2009
Non – Determinism

• Current Shared Memory Multicore and Multiprocessor systems → multithreaded application – same inputs can produce different outputs. (threads can interleave their memory and I/O operations differently each time !)

• Result : Change in program behaviour in each execution

• Debugging and Testing problems. Makes software development process complicated.

• Case for a fully deterministic shared memory multiprocessing : DMP
Defining Deterministic Parallel Execution

• Execute multiple threads that communicate via shared memory and produce same output for the same input.

• Same global interleaving of instructions.

• All communication between threads must be same for each execution.

• Carefully control the behaviour of Load and Store operations that cause inter thread communication.
Parallel execution

Examples of communication-equivalent global interleavings
Sources of Nondeterminism

• **Software sources**: Other concurrent processes competing for resources; state of memory pages, power savings mode, disc and I/O buffers, state of global registers in the OS.

• **Hardware sources**: No of non-ISA visible components that vary from run to run: architectural structures like state of any caches, predictor tables and bus priority controllers. Environmental factors.

Footnote: Today’s hardware and software are not built to behave deterministically.
Actually measured.
Enforcing DMP

DMP Serial:

- Allow only one processor at a time to access memory in deterministic order.

- Deterministic Serialisation of a parallel execution.

- Memory Access Token method.

- Need to Recover Parallelism for acceptable performance
Quantum

(a) Parallel execution
(b) Deterministic serialized execution at a fine-grain
(c) Deterministic serialized execution at a coarse-grain
DMP-ShTab:

- Threads do not communicate all the time. Until they communicate: full on parallel (& between communication)

- Deterministic Serialisation again when threads communicate. Each quantum → broken into
  a) communication free prefix (I’ll exec with other quanta) &
  b) suffix (first point of communication) executes serially.

- Mechanism for inter-thread communication.

- Sharing table.
Initially A is private in P0

Initially A is shared

P0

P1

P1 deterministically reading another thread's private data

P0 deterministically writing to shared data
thread communication

1. If A is owned by t:
   - Proceed with access
2. If A is shared:
   - If read:
     - Set A private and owned by thread t
     - If write:
       - Set A shared
   - t waits for token and for all other threads to be blocked
3. If A is not shared:
   - t waits for token and for all other threads to be blocked
Support for TM: DMP-TM and DMP-TMFwd

- Encapsulate each quantum inside a transaction, make it appear to execute atomically and in isolation.

- Mechanism to form quanta deterministically, to enforce a deterministic commit order.

- Speculative concurrent runs until overlapping memory accesses (violation of original Det. Serialisation. of memory operations).

- TM-Fwd allows uncommitted (speculative) data forwarding between quanta → performance enhanced.
memory operation

→ deterministic token passing

N deterministic order

atomic quantum

→ uncommitted value flow

(a) Pure TM

(b) TM w/ uncommitted value flow
We allow a quantum to fetch speculative data from another uncommitted quantum earlier in det. total order.

If a quantum that provided data to another quantum is squashed, all subsequent quanta must also be squashed.
Better Quantum Building

QB Count
QB SyncFollow
QB Sharing
QB SyncSharing

(a) Regular quantum break

(b) Better quantum break

- memory operation
- deterministic order
- deterministic token passing
- atomic quantum
Implementation

• Primarily requires mechanisms to:
  -- build quanta
  -- guarantee deterministic serialisation.

Software vs Hardware Trade-Off.

• Hw-DMP Serial: Support for token (multiple) passing.
• Hw-DMP ShTab: Sharing table Data Structure.
• Hw-DMP-TM and Hw-DMP-TMFwd: A Mechanism to enforce specific transaction commit order, TM-Fwd needs speculative data flow support – making the coherence protocol aware. (TLS).
Software-only implementation,

- Using a compiler or a binary rewrite infrastructure.

- Compiler builds quanta – tracks dynamic instruction count in the Control Flow Graph by sparsely inserting code.

- SwDMP-Serial implements deterministic token as a queuing clock. For DM-SHTab, compiler causes every load and store to call back to the run time system that implements the logic discussed earlier.
Experimental Setup

- Use of SPLASH2 and PARSEC benchmark suites.

- Some infrastructure limitations. Simulations run on a dual Intel Xeon quad-core 64 bit processor 2.8 GHz machine.

- Hw-DMP : a) Simulator to assess performance written using PIN. Includes quantum building, memory conflict, squashes due to speculation support. b) Averaging of results over multiple times for real-time-like results.

- Sw-DMP : Performance evaluated using LLVMv2.2 Compiler pass.
Performance Evaluations

Runtime overheads with 4, 8 and 16 threads.
Performance of 2,000(2), 10,000(X) and 100,000(C) instruction quanta, relative to 1000 instruction quanta
Performance of QB-Sharing(s), QB-SyncFollow(sf) and QBSyncSharing (ss) quantum builders, relative to QBCount, with 1,000-insn quanta.
Performance of quantum building schemes, relative to QB-Count, with 10,000-insn quanta.
Runtime of Sw-DMPSHTab relative to nondeterministic execution.
Inferences

• Deterministic execution possible with little or no performance degradation.

• DMP-Serial has a GM slowdown of 6.5 X on 16 threads.

• DMP-ShTab -- slowdown 15%

• HwDMP-TM – reduction in slowdown to 10%

• HwDMP-TMFwd – slowdown less than 8%

• Software solutions : Cost effective Deterministic execution, suitable for debugging.
Other Issues

• Inferences show that speculation improves performance, but wastes energy, and increases complexity of system design.

• Trade-off: DMP Serial, DMP-ShTab and DMP TM can co-exist. Switch at the end of quanta (boundary). Decision can be made based on code!

• Hybrid system: Software + Hardware. Eg: Hybrid DMP – TM → Modest hardware TM support, use of software for quantum building and deterministic ordering. Minimises Performance cost.
Other Issues : More Non-Determinism

• Parallel programs can use OS to communicate between threads. This communication must be made deterministic.
  ---- Execute OS code deterministically
  ---- Layer to provide synchronisation btw OS and app.

• Operating System calls are designed to allow non-determinism. Eg. Read. Solns : set a rule that read will always return maximum amount of data requested.

Related work, References

- Deterministic parallel programming models: StreamIt. Implicitly parallel languages: Jade → Domain Specific.

- **Deterministic Replay**: A record of the log of the ordering of events during parallel execution, for debugging later. Several software Replay systems. High overhead.

- Hardware Replay systems. Eg: Strata, ReRun, DeLorean. ReRun: records hardware memory race (records execution periods without memory communication)
In vein with DMP

- DeLorean: Instructions are executed as blocks and commit order of instructions is recorded. (Not each instruction).

- Uses pre-defined commit ordering to reduce memory ordering log. That is: it reduces log size by controlling Non-determinism. But DMP needs no logging. It makes execution totally deterministic. No need for REPLAY.

- DMP quanta vs DeLorean chunk?

- Thread Level Speculation (TLS)?
Conclusion

• The case for Deterministic Execution.

• Achievement of the same using DMP and variations.

• Proof of comparable performance with parallel Nondeterministic systems. Makes debugging easier.

• Stresses the need for “determinism in the field”

• Is writing, debugging and deploying parallel code as difficult as it was at the beginning of this paper ???? ????