Understanding Transactional Memory

João Lourenço
Bruno Teixeira
Ricardo Dias
Ricardo Pinto
Diogo Sousa
Vasco Pessanha

Center for Informatics and Information Technologies
Computer Science Department
New University of Lisboa, Portugal

Joao.Lourenco@di.fct.unl.pt
Agenda

1. Software Transactional Memory: what, why and how?

2. Detection of dataraces in Transactional Memory

3. Visualization of Transactional Memory computations

4. Final notes
The problem

- Multicores are here to stay
  - Moore’s law
  - Power consumption
  - Home computers are multicore now
    - Soon will be manycore
The problem

- Computers are data processing tools
  - Data may be scattered among different computing units
  - A set of data manipulations may be inter-related and inter-dependent
  - The isolated change of a data item may leave the overall system in an inconsistent state
The problem

- Parallel programming is hard
  - Multiple (asynchronous) control flows
  - Concurrency control
    - Synchronization
    - Contention management
The problem

- For developing parallel code we need
  - Parallel programming skills
    - Programming languages constructs
    - System / run-time support
    - Techniques and tools for (parallel) program development
• Blocking methodologies
  • Locks
  • Condition variables
  • Semaphores

• Non-blocking algorithms
  • e.g., java.util.concurrent
Locks: Insert in a Double Linked List
Coarse grain locks

```java
public synchronized void insertNode (node, precedingNode) {
    node.prec = precedingNode;
    node.succ = precedingNode.succ;
    precedingNode.succ.prec = node;
    precedingNode.succ = node;
}
```
Fine grain locks

public void insertNode (node, precedingNode) {
    synchronized (precedingNode) {
        synchronized (precedingNode.succ) {
            node.prec = precedingNode;
            node.succ = precedingNode.succ;
            precedingNode.succ.prec = node;
            precedingNode.succ = node;
        }
    }
}
Fine grain locks lead to deadlocks

Fine grain locks

```java
synchronized (precedingNode) {
    synchronized (precedingNode.succ) {
        ...
    }
}
```

Deadlock

```java
synchronized (precedingNode.succ) {
    synchronized (precedingNode) {
        ...
    }
}
```
Problem: locking granularity

- Coarse grain locks
  - Threads lock each other out
  - Lower concurrency

- Fine grain locks
  - Higher overhead
  - Risk of deadlocks
  - Lower maintainability
Example: bank account transfer

- Transfer 50€ from account A to account B

1. read (A);
2. $A = A - 50$;
3. write (A);
4. Read (B);
5. $B = B + 50$;
6. write (B);
Example — Atomicity

- Transfer 50€ from account A to account B

1. read (A);
2. A = A – 50;
3. write (A);
4. Read (B);
5. B = B + 50;
6. write (B);

**Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system must ensure that updates are not reflected in the database.
Example — Consistency

- Transfer 50€ from account A to account B
  1. read (A);
  2. A = A – 50;
  3. write (A);
  4. Read (B);
  5. B = B + 50;
  6. write (B);

**Consistency requirement** — the sum of A and B is unchanged by the execution of the transaction
Example — Isolation

- Transfer 50€ from account A to account B

1. read (A);
2. A = A – 50;
3. write (A);
4. Read (B);
5. B = B + 50;
6. write (B);

**Isolation requirement** — no other successful transaction can see an intermediate state of this transaction (where the temporary sum of the accounts is less than it should)
Example — Durability

- Transfer 50€ from account A to account B

1. read (A);
2. A = A – 50;
3. write (A);
4. Read (B);
5. B = B + 50;
6. write (B);

**Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the 50€ has taken place), the updates to the database by the transaction must persist despite failures
Properties

- **Consistency** — the transaction must obey legal protocols

- **Atomicity** — it either happens or it does not; either all are bound by the contract or none are

- **Isolation** — transactions must behave as if executed alone in the system

- **Durability** — once a transaction is committed, it cannot be abrogated
Fine grain locks

public void insertNode (node, precedingNode) {
    synchronized (precedingNode) {
        synchronized (precedingNode.succ) {
            node.prec = precedingNode;
            node.succ = precedingNode.succ;
            precedingNode.succ.prec = node;
            precedingNode.succ = node;
        }
    }
}
Transactional memory

A possible solution


```java
public void insertNode (node, precedingNode) {
    atomic {
        //means "transaction"
        node.prec = precedingNode;
        node.succ = precedingNode.succ;
        precedingNode.succ.prec = node;
        precedingNode.succ = node;
    }
}
```
Transactional memory

- Transactional operations
  - Established in the domain of databases
  - Easy to understand

- Transaction
  - Series of read and write operations
  - Executed at least atomically and isolated (AI from ACID properties)
  - Performance is an issue

- On conflict
  - Transaction is repeated
Transactional memory

- Advantages
  - Developer worries less about parallelism
  - Code is marked as atomic/isolated
  - Code is generated and run-time ensure the properties

- Disadvantages
  - Not always ideal
  - Successive collisions ➔ repetitive rollbacks
  - Memory transactions cannot contain certain operations (e.g., I/O operations)
Expectations

- Preserving real-time ordering
  - The conceptual point in time a transaction appear to execute lies within its lifespan
    - i.e., transactions do no execute “in the past” neither “in the future”

- Precluding inconsistent views
  - Assumption 1: transactions that access an inconsistent memory state will generate an exception
  - Assumption 2: transactions that generate exceptions are aborted
Types of Actions in STM

- Non-transactional operations:
  - Action whose effects are observed by an entity external to the system
    - Killing a process
    - User seeing a printf
  - Irreversible and cannot be undone
Types of Actions in STM

- Transactional operations:
  - Pure transactional
    - Storing a value in memory
    - Execute a gettimeofday() system call
  - Revertible / Compensable
    - Memory management operations
    - Execute a lseek() system call
    - Appending data to a file
    - Execute the creat() system call
Non-Transactional Operations in STM

- **Possible approaches:**
  - Don’t care
    - Leave to the programmer the responsibility to avoid or manage its use in TM blocks
    - Common approach for library-based TM frameworks
  - **Do not allow**
    - The programming language type system do not allow its use inside TM blocks
    - Approach in Concurrent Haskell
      - IO operations are typified and statically verified by the compiler
        - e.g., putChar :: Char -> IO()
Non-Transactional Operations in STM

- Possible approaches:
  - Enter a non-speculative mode
    - Enforce that transactions execution NTO will never abort
    - May use an optimistic approach, assuming transactions will not execute NTO, otherwise...
  - Wait for all other concurrent transactions to finish
  - Restart and run the transaction that will execute NTOs as the only transaction in the system
  - Once finished, resume the normal operation
Revertible / Compensable Operations in STM

- Possible approaches:
  - Defer
    - Some operations may/must be postponed to the end of the atomic block
  - Compensate
    - Some operations are not revertible, but may be compensated later without compromising the application semantics
Handling Revertible Operations in STM

- **Deferred operations** can be supported by transaction handlers

- Handlers extend the transaction life cycle with new states
  - Example for memory transactions in next slide
Memory Transaction States

BEGIN → active → partially committed → committed → END
BEGIN → active → aborted → END
```c
void *my_malloc(...) {
    void *m = malloc (...);
    register_pre_abort(free, m);
}
```

```c
#define my_free (m) \
    register_pos_commit (free, m);
```

---

**Memory Transaction States**

- **BEGIN**
- **active**
- **pre-abort**
- **free local malloc()**
- **partially committed**
- **prepare-commit**
- **pre-commit**
- **committed**
- **aborted**
- **pos-abort**
- **END**

**Transaction States**

**Handler System States**

**postponed free()**

**commit**

**pre-commit**

**commit**

**post-commit**
An Handler System

TWO PHASE COMMIT

prepare-commit
pre-commit

pos-commit

pre-abort

pos-abort

START_TX

BODY_TX

COMMIT_TX

ABORT_TX
```c
void TxDBStart(...) {
    TxStart();
    register_pre_abort(...);
    register_prepare_commit(...);
    register_pre_commit(...);
}
```
A Static Approach for Detecting Concurrency Anomalies in Transactional Memory
Problem Statement

- Concurrency errors have always been a problem
  - Dataraces
  - Deadlocks
  - Priority inversion
  - Convoying
  - ...

- There are still errors observed in TM
  - Low-level dataraces
  - High-level dataraces

No tools for TM

Our goal is to statically detect these errors
Low-level Dataraces (LLDR)

```java
private boolean hasSpaceLeft() {
    return (list.size() < MAX_SIZE);
}

private void store(Object obj) {
    list.add(obj);
}

public void attemptToStore(Object obj) {
    if (hasSpaceLeft()) {
        store(obj);
    }
}
```
Low-level Dataraces (LLDR)

```java
private boolean hasSpaceLeft() {
    return list.size() < MAX_SIZE;
}

private void store(Object obj) {
    list.add(obj);
}

public void attemptToStore(Object obj) {
    if (hasSpaceLeft()) {
        store(obj);
    }
}
```

Low-level datarace: `list` is read and concurrently updated.
Low-level Dataraces (LLDR)

```java
private boolean hasSpaceLeft() {
    synchronized {
        return (list.size() < MAX_SIZE);
    }
}

private void store(Object obj) {
    synchronized {
        list.add(obj);
    }
}

public void attemptToStore(Object obj) {
    if (hasSpaceLeft()) {
        store(obj);
    }
}
```
private boolean hasSpaceLeft() {
    synchronized {
        return (list.size() < MAX_SIZE);
    }
}

private void store(Object obj) {
    synchronized {
        list.add(obj);
    }
}

public void attemptToStore(Object obj) {
    if (hasSpaceLeft()) {
        // list may become full
        store(obj);
    }
}
High-level Dataraces (HLDR)

```java
private boolean hasSpaceLeft() {
    synchronized {
        return (list.size() < MAX_SIZE);
    }
}

private void store(Object obj) {
    synchronized {
        list.add(obj);
    }
}

public void attemptToStore(Object obj) {
    if (hasSpaceLeft()) {
        // list may become full
        store(obj);
    }
}
```

High-level datarace: accesses are synchronized but list can overflow
LLDR Detection

- Automatically convert TM to lock based programs
- New program is processed by a datarace detector
- Detected dataraces are also present in TM program

Control with TM

<table>
<thead>
<tr>
<th>Access 1</th>
<th>Access 2</th>
<th>DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
<td>DR</td>
</tr>
<tr>
<td>T</td>
<td>Ø</td>
<td>DR</td>
</tr>
<tr>
<td>Ø</td>
<td>T</td>
<td>DR</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

Control with Locks

<table>
<thead>
<tr>
<th>Access 1</th>
<th>Access 2</th>
<th>DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
<td>DR</td>
</tr>
<tr>
<td>L1</td>
<td>Ø</td>
<td>DR</td>
</tr>
<tr>
<td>Ø</td>
<td>L1</td>
<td>DR</td>
</tr>
<tr>
<td>L1</td>
<td>L2</td>
<td>DR</td>
</tr>
</tbody>
</table>

Ø : Unprotected access
L : Lock-guarded access
T : Transactional access
LLDR Detection

Automatic Transformer (Polyglot based tool)

TM-based Program

Lock-based (single lock) Program

Datarace Detector (JChord)

Datarace Report

LLDR Detector
Example

```java
private boolean hasSpaceLeft() {
    atomic { return (list.size() < MAX_SIZE); }
}
private void store(Object obj) {
    atomic { list.add(obj); }
}

public static Object GLOCK;

private boolean hasSpaceLeft() {
    synchronized (GLOCK) { return (list.size() < MAX_SIZE); }
}
private void store(Object obj) {
    synchronized (GLOCK) { list.add(obj); }
}
```
Validation: Lee-TM

- Renowed benchmark
- 2 versions (lock- and TM-based)

- Ran JChord in lock version
  - 52 dataraces – \textit{nolocks} and \textit{wrong-locks}

- Ran LLDR detector in TM version
  - 48 dataraces – accesses outside transactional blocks
High-level Dataraces (HLDR)

```java
private boolean hasSpaceLeft() {
    synchronized {
        return (list.size() < MAX_SIZE);
    }
}

private void store(Object obj) {
    synchronized {
        list.add(obj);
    }
}

public void attemptToStore(Object obj) {
    if (hasSpaceLeft()) {
        // list may become full
        store(obj);
    }
}
```

**High-level Anomaly:**
List could overflow, although accesses are synchronized.
HLDR Detection

TM-based Program

AJEX Parser (Polyglot extension)

AST

Symbolic Executor/Tracer

Traces

Pattern Matcher

Anomaly Reports

HLDR Detector
private boolean hasSpaceLeft() {
    atomic {
        return (list.size() < MAX_SIZE);
    }
}

private void store(Object obj) {
    atomic {
        list.add(obj);
    }
}

public void attemptToStore(Object obj) {
    if (hasSpaceLeft())
        store(obj);
}

public void run() {
    attemptToStore("I’m a misbehaving program.");
}
Tracer (Example)

- **hasSpaceLeft()**
  - **ATOMIC**
  - R: list
  - R: MAX_SIZE

- **attemptToStore()**
  - **CALL**: hasSpaceLeft()
  - **IF**
    - **CALL**: store()
  - **END**

- **store()**
  - **ATOMIC**
  - R: list
  - W: list

- **run()**
  - **CALL**: attemptToStore()
Tracer (Example)

```
hasSpaceLeft()

ATOMIC
R: list
R: MAX_SIZE

CALL: hasSpaceLeft()

IF

CALL: store()

END

store()

ATOMIC
R: list
W: list

CALL: attemptToStore()

run()
```
Tracer (Example)

```
ATOMIC
R: list
R: MAX_SIZE

ATOMIC
R: list
W: list

CALL: hasSpaceLeft()

IF
CALL: store()

END
```
Tracer (Example)

```
ATOMIC
R: list
W: list

run()

IF

CALL: store
()

END
```
Tracer (Example)

Trace 1

START

ATOMIC
R: list
R: MAX_SIZE

ATOMIC
R: list
W: list

END

Trace 2

START

ATOMIC
R: list
R: MAX_SIZE

ATOMIC
R: list
R: MAX_SIZE

ATOMIC
R: list
W: list

1 END

END
Tracing: Calls and Flow Control

- **Method calls**
  - Target method is expanded (inlined)

- **If statement**
  - Both branches are considered, but independently

- **Loops**
  - 0, 1, and 2 iterations are considered

- **Recursive calls**
  - Method is expanded twice, avoid infinite expansions but yield all possible anomalies
HLDR Detection

TM-based Program

Anomaly Reports

HLDR Detector

AJEX Parser (Polyglot extension)

AST

Symbolic Executor/Tracer

Traces

Pattern Matcher
Anomaly Patterns

- Relation between transactions may be inferred by data access patterns

- Certain access patterns may indicate anomalies
Anomaly Patterns

Transaction 1:
Thread 1: Read(x) -> Write(x) -> Read(x)
Thread 2: Write(x,y) -> Read(x,y)
Anomaly: RwR

Transaction 2:
Thread 1: Write(x) -> Write(x)
Thread 2: Read(x,y) -> Write(x)
Anomaly: WrW

Transaction 3:
Thread 1: Read(x) -> Write(x)
Thread 2: Write(x)
Anomaly: RwW
Anomaly Pattern: list is retrieved, and updated in another transaction
Validation

Well-known Lock-based Buggy Programs

Hand Rewriting

TM-based Buggy Programs

Anomaly Reports

TM HLDR Detection (Polyglot)
Coordinates’03

//Thread 3
public void run() {
    double x = c.getX(); //atomic
    double y = c.getY(); //atomic
    // use x and y...
}

//Thread 1
public void run() {
    c.setXY(...); //atomic
}

//Thread 4
public void run() {
    Coord d4 = c.getXY();
    double x4 = d4.getX();
    double y4 = d4.getY();
}

Anomaly: RwR
Coordinates could be from different states
False Positives
pointer analysis required
```java
public class Counter {
    int i;
    int inc(int a) {
        atomic {
            i = i + a;
            return i;
        }
    }
}

// Thread code
public void run() {
    int i = c.inc(0); // get i
    // value could have changed
    c.inc(i); // increment by i
}
```

Anomaly: **RwW**

Updates could be lost

No False Positives
## Results

<table>
<thead>
<tr>
<th>Test</th>
<th>Total Anomalies</th>
<th>Total Warnings</th>
<th>Correct Warnings</th>
<th>False Warnings</th>
<th>Missed Anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Coord’03</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Local Var</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NASA</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Coord’04</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Buffer</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Dbl-Check</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>StringBuffer</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Account</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Over-report</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Under-report</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Alloc Vector</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Knight Moves</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>12</strong></td>
<td><strong>33</strong></td>
<td><strong>10</strong></td>
<td><strong>23</strong></td>
<td><strong>2</strong></td>
</tr>
</tbody>
</table>
## Results

**Summary for 14 tests performed**

<table>
<thead>
<tr>
<th>Total Anomalies</th>
<th>Total Warnings</th>
<th>Correct Warnings</th>
<th>False Warnings</th>
<th>Missed Anomalies</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>33</td>
<td>10</td>
<td>23</td>
<td>2</td>
</tr>
</tbody>
</table>

**70% of total results**
- 3 RwR
- 3 WrW
- 6 RwW

**Due to Unavailable source-code**
- 1 RwR
- 1 RwW

- 5 redundant reads
- 6 pointer analysis
- 10 pattern refinement
- 2 permanent
Monitoring and Visualizing Transactional Memory Programs
Message Passing...
“how did we used to do?”
Shared Memory...

"how did we used to do?"

Clear lack of tools
Transactional Memory...
“how did we used to do?”

Total lack of tools
Transactional Memory...
“how do we do now?”
Requirements for Tracing in JTraceView

- Keep hardware requirements
  - Different hardware may mask some behaviors and trigger new ones

- Log only well defined time-slots
  - When to start to collect events and for how long
  - Huge log files

- Keep the global application behavior
  - Same behavior pattern
  - Low and constant overhead
  - No additional synchronizations
Our Proposal for Tracing in JTraceView

- Keep events in separate buffers
  - One buffer per thread
  - Avoids additional synchronizations

- Use RDTSC (or similar) registers as a global clock
  - Clock drifting can be a problem
    - In many cases it is not
  - Provides info on real execution time for transactions

- Keep global application behavior
Behavior of Memory Transactions

- Specification:
  \[ \text{Tx\_Start} \ (\text{Tx\_Read} \ | \ \text{Tx\_Write})^* \ (\text{Tx\_Abort} \ | \ \text{Tx\_Commit}) \]

- Run-time:
  \[ \text{Tx\_Start} \ (\text{Tx\_Read} \ | \ \text{Tx\_Write})^* \ (\text{Tx\_Abort\_Read} \ | \ \text{Tx\_Abort\_Write} \ | \ \text{Tx\_Abort\_Commit} \ | \ \text{Tx\_Abort\_User} \ | \ \text{Tx\_Commit}) \]
Events for TM Tracing

**All Events**

- **timestamp**
  - When?

- **eventID**
  - What?

- **threadID**
  - Where?

- **transID**
  - Whom?

**Tx_Read | Tx_Write**

- **varID**
  - Memory location
  
  0x84ef2e8

**Tx_Abort**

- **evType**
  - Why?

  User | Abort | Other
Keeping App Behavior

- **Linked List**

  - Unmonitored
  - JTraceView tracing
  - Naïve tracing

- **Red-Black Tree**

  - Unmonitored
  - JTraceView tracing
  - Naïve tracing
<table>
<thead>
<tr>
<th>Address (HEX)</th>
<th>Function</th>
<th>Process</th>
<th>Value (HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>357682676772344</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84ef2e8</td>
</tr>
<tr>
<td>357682676773517</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84f0d50</td>
</tr>
<tr>
<td>357682676774549</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84ef2a8</td>
</tr>
<tr>
<td>357682676775661</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676776340</td>
<td>tx_read</td>
<td>T2</td>
<td>0x128</td>
</tr>
<tr>
<td>357682676777018</td>
<td>tx_write</td>
<td>T2</td>
<td>0x127</td>
</tr>
<tr>
<td>357682676777932</td>
<td>tx_read</td>
<td>T2</td>
<td>0x18c</td>
</tr>
<tr>
<td>357682676778553</td>
<td>tx_write</td>
<td>T2</td>
<td>0x18b</td>
</tr>
<tr>
<td>357682676779150</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676780071</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676780703</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84f0d50</td>
</tr>
<tr>
<td>357682676781647</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676782276</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84f0d50</td>
</tr>
<tr>
<td>357682676782893</td>
<td>tx_write</td>
<td>T2</td>
<td>0x84ef2b8</td>
</tr>
<tr>
<td>357682676783823</td>
<td>tx_write</td>
<td>T2</td>
<td>0x84ef2b8</td>
</tr>
<tr>
<td>357682676784422</td>
<td>tx_commit</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676785640</td>
<td>tx_start</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676791481</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676792077</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676793013</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676793651</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676794275</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676795214</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84ef70</td>
</tr>
<tr>
<td>357682676795835</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84ef70</td>
</tr>
<tr>
<td>357682676796464</td>
<td>tx_read</td>
<td>T2</td>
<td>0x88</td>
</tr>
<tr>
<td>357682676797411</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84f0e0b</td>
</tr>
<tr>
<td>357682676798039</td>
<td>tx_read</td>
<td>T2</td>
<td>0x88</td>
</tr>
<tr>
<td>357682676799270</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84f0e0b</td>
</tr>
<tr>
<td>357682676800314</td>
<td>tx_read</td>
<td>T2</td>
<td>0x86</td>
</tr>
<tr>
<td>357682676801265</td>
<td>tx_write</td>
<td>T2</td>
<td>0x84f0e0b</td>
</tr>
<tr>
<td>357682676802315</td>
<td>tx_write</td>
<td>T2</td>
<td>0x87</td>
</tr>
<tr>
<td>357682676802934</td>
<td>tx_write</td>
<td>T2</td>
<td>0x80</td>
</tr>
<tr>
<td>357682676803569</td>
<td>tx_read</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676804519</td>
<td>tx_commit</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676807718</td>
<td>tx_start</td>
<td>T2</td>
<td>0x0</td>
</tr>
<tr>
<td>357682676808381</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84dd600</td>
</tr>
<tr>
<td>357682676808920</td>
<td>tx_read</td>
<td>T2</td>
<td>0x3</td>
</tr>
<tr>
<td>357682676809561</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84eb408</td>
</tr>
<tr>
<td>357682676810472</td>
<td>tx_read</td>
<td>T2</td>
<td>0x78</td>
</tr>
<tr>
<td>357682676811097</td>
<td>tx_read</td>
<td>T2</td>
<td>0x84f0fc0</td>
</tr>
</tbody>
</table>

**RB-Tree in CTL generates 0.5 KB per processor per milisecond**

Unmanageable by common mortals

Even when converted to text
JTraceView: Visualization Tool for TM

- Trace file processor
- Trace file analyzers
- Monitoring Layer
- Visualization plugins

JTraceView
JTraceView: Visualization Tool for TM

- Trace file processor
- Trace file analyzers
- Monitoring Layer
- Visualization plugins

JTraceView

Individual Transaction Percentages

Haifa Verification Conference, October 4, 2010 / 84
Visualization Plugins

Transaction Commit/Abort Percentages

Abort Percentages

Transaction Retry Rate

MemCells

Individual Transactions

Haifa Verification Conference, October 4, 2010 / 85
Example

Transaction Retry Rate

LL, 8,
20%, 20%, 60%,
50

Transaction Retry Rate

RB, 8,
45%, 45%, 10%,
50
Stanford Transactional Applications for Multi-Processing

- Collection of applications for TM benchmarking
- Includes both sequential and parallel code sections
- Eight different applications
  - Six were ported to Java (DeuceSTM framework)

http://stamp.stanford.edu
Signature-based network intrusion detection systems (NIDS) scan network packets for matches against a known set of intrusion signatures.

Properties:
- Short transactions length
- Medium read/write sets
- High contention
- Medium transaction time
Trace Data

- Collected in a Sun Fire x4600
  - 8 x AMD Opteron Model 8220 dual-core processor @ 2.8GHz
  - 32 GB DDR2-667 memory
  - Linux Debian lenny

- Test Conditions
  - TM algorithm: TL2
  - Run time: 20 secs
  - Trace File Size: 37MB
  - Total transactions: 20814
  - Total commit: 14254
  - Total abort: 6560
STAMP: Intruder

- Includes three transactional code blocks
  - Code 0 —
    `atomicGetPacket (streamPtr);`
    Pop data from a queue
  - Code 1 —
    `atomicProcess (decoderPtr, packetPtr);`
    Assemble packet segments and decode them
  - Code 2 —
    `atomicGetComplete (decoderPtr, decodedFlowId);`
    Get decoded data of the packet
Transactional Code Blocks

Individual Transaction Percentages

- Code 0: 33%
- Code 1: 33%
- Code 2: 33%
Read / Write Rates

Read/Write Transaction Rates

<table>
<thead>
<tr>
<th>Code</th>
<th>Read Transactions</th>
<th>Write Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>80,466</td>
<td>19,534</td>
</tr>
<tr>
<td>1</td>
<td>80,792</td>
<td>19,208</td>
</tr>
<tr>
<td>2</td>
<td>85,855</td>
<td>14,145</td>
</tr>
</tbody>
</table>
Abort Types

Abort Percentages

- Read 69%
- Commit 31%
- User 0%
- Write 0%
Abort — False Conflicts

Abort False Positives

- True Conflicts 88%
- Doomed Conflicts 12%
- False Conflicts 0%
Commits/Aborts of Threads per time
Transaction Abort / Retry

![Graph showing Transaction Retry of each code per time](image)
Duration of Transactions
Thread timeline

![Thread timeline diagram](image)
Thread timeline
Concluding Remarks

- Transactional Memory is an appealing paradigm for specifying concurrent programs

- We need software development tools specifically targeting TM
  - Traditional tools do not apply to TM

- Transactional Memory programs may also exhibit data races
  - Low-level data races: only in weak-isolation
  - High-level data races: always
Concluding Remarks

- It is possible to detect LLDR’s in TM by analyzing “equivalent” lock-based programs

- Most HLDR’s occur when a transaction runs between two consecutive transactions of another thread

  - The three patterns identified capture most of the anomalies referenced in the literature

  - Good detection rates for a set of well known examples from different sources
Concluding Remarks

- JTraceView includes a low intrusion monitoring layer
- Deals with the very high frequency of events and huge logs
- The huge logs can be visually synthesized
  - Statistical charts
  - Time-space diagram
- Time-space view may also help in debugging
Want to try it yourself?

- Get:
  - TM framework instrumented to generate logs (tracing data)
  - JTraceView to visualize tracing data

- From:
  - [http://asc.di.fct.unl.pt/trxsy](http://asc.di.fct.unl.pt/trxsy)
    - Check the prototypes page

- When:
  - From October 16th
The End...

Have fun!

😊