Transactional Memory

Prof. Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Tech

(Adapted from Stanford TCC group and MIT SuperTech Group)
Motivation

• Uniprocessor Systems
  – Frequency
  – Power consumption
  – Wire delay limits scalability
  – Design complexity vs. verification effort
  – Where is ILP?

• Support for multiprocessor or multicore systems
  – Replicate small, simple cores, design is scalable
  – Faster design turnaround time, Time to market
  – Exploit TLP, in addition to ILP within each core
  – But now we have new problems
Parallel Software Problems

- Parallel systems are often programmed with
  - Synchronization through **barriers**
  - Shared objects access control through **locks**

- Lock granularity and organization must balance performance and correctness
  - Coarse-grain locking: Lock contention
  - Fine-grain locking: Extra overhead
  - Must be careful to avoid deadlocks or data races
  - Must be careful not to leave anything unprotected for correctness

- Performance tuning is not intuitive
  - Performance bottlenecks are related to low level events
    - E.g. false sharing, coherence misses
  - Feedback is often indirect (cache lines, rather than variables)
Parallel Hardware Complexity (TCC’s view)

• Cache coherence protocols are complex
  – Must track ownership of cache lines
  – Difficult to implement and verify all corner cases

• Consistency protocols are complex
  – Must provide rules to correctly order individual loads/stores
  – Difficult for both hardware and software

• Current protocols rely on low latency, not bandwidth
  – Critical short control messages on ownership transfers
  – Latency of short messages unlikely to scale well in the future
  – Bandwidth is likely to scale much better
    • High speed interchip connections
    • Multicore (CMP) = on-chip bandwidth
What do we want?

• A shared memory system with
  – A simple, easy programming model (unlike message passing)
  – A simple, low-complexity hardware implementation (unlike shared memory)
  – Good performance
Lock Freedom

• Why lock is bad?
• Common problems in conventional locking mechanisms in concurrent systems
  – Priority inversion: When low-priority process is preempted while holding a lock needed by a high-priority process
  – Convoying: When a process holding a lock is de-scheduled (e.g. page fault, no more quantum), no forward progress for other processes capable of running
  – Deadlock (or Livelock): Processes attempt to lock the same set of objects in different orders (could be bugs by programmers)
• Error-prone
Using Transactions

• What is a transaction?
  – A sequence of instructions that is guaranteed to execute and complete only as an atomic unit

    Begin Transaction
    Inst #1
    Inst #2
    Inst #3
    ...
    End Transaction

  – Satisfy the following properties
    • Serializability: Transactions appear to execute serially.
    • Atomicity (or Failure-Atomicity): A transaction either
      – commits changes when complete, visible to all; or
      – aborts, discarding changes (will retry again)
TCC (Stanford) [ISCA 2004]

• Transactional Coherence and Consistency
• Programmer-defined groups of instructions within a program

  Begin Transaction
  Inst #1
  Inst #2
  Inst #3
  ...
  End Transaction

  Start Buffering Results

  Commit Results Now

• Only commit machine state at the end of each transaction
  – Each must update machine state atomically, all at once
  – To other processors, all instructions within one transaction appear to execute only when the transaction commits
  – These commits impose an order on how processors may modify machine state
Transaction Code Example

- MIT LTM instruction set

```assembly
xstart:
    XBEGIN on_abort
    lw    r1, 0(r2)
    addi  r1, r1, 1
    ...
    XEND
    ...
    on_abort:
    ...
    // back off
    j xstart  // retry
```
Transactional Memory

- Transactions **appear** to execute in commit order
  - Flow (RAW) dependency cause transaction violation and restart

![Diagram showing transactional memory with transactions A, B, and C. Transaction A is committed successfully. Transaction B is blocked due to a flow dependency and re-executes with new data. Transaction C is marked with a violation symbol.](image)
Transactional Memory

- Output and Anti-dependencies are automatically handled
  - WAW are handled by writing buffers only in commit order (think about sequential consistency)
Transactional Memory

- Output and Anti-dependencies are automatically handled
  - WAW are handled by writing buffers only in commit order
  - WAR are handled by keeping all writes private until commit
TCC System

• Similar to prior thread-level speculation (TLS) techniques
  – CMU Stampede
  – Stanford Hydra
  – Wisconsin Multiscalar
  – UIUC speculative multithreading CMP

• Loosely coupled TLS system

• Completely eliminates conventional cache coherence and consistency models
  – No MESI-style cache coherence protocol

• But require new hardware support
The TCC Cycle

- Transactions run in a cycle
- Speculatively execute code and buffer
- Wait for commit permission
  - **Phase** provides synchronization, if necessary
  - Arbitrate with other processors
- Commit stores together (as a packet)
  - Provides a well-defined write ordering
  - Can invalidate or update other caches
  - Large packet utilizes bandwidth effectively
- And repeat
Advantages of TCC

• Trades bandwidth for simplicity and latency tolerance
  – Easier to build
  – Not dependent on timing/latency of loads and stores

• Transactions eliminate locks
  – Transactions are inherently atomic
  – Catches most common parallel programming errors

• Shared memory consistency is simplified
  – Conventional model sequences individual loads and stores
  – Now only have hardware sequence transaction commits

• Shared memory coherence is simplified
  – Processors may have copies of cache lines in any state (no MESI !)
  – Commit order implies an ownership sequence
How to Use TCC

• Divide code into potentially parallel tasks
  – Usually loop iterations
  – For initial division, tasks = transactions
    • But can be subdivided up or grouped to match HW limits (buffering)
  – Similar to threading in conventional parallel programming, but:
    • We do not have to verify parallelism in advance
    • Locking is handled automatically
    • Easier to get parallel programs running correctly

• Programmer then orders transactions as necessary
  – Ordering techniques implemented using phase number
  – Deadlock-free (At least one transaction is the oldest one)
  – Livelock-free (watchdog HW can easily insert barriers anywhere)
How to Use TCC

• Three common ordering scenarios
  – Unordered for purely parallel tasks
  – Fully ordered to specify **sequential** task (algorithm level)
  – Partially ordered to insert synchronization like barriers
Basic TCC Transaction Control Control Bits

• In each local cache
  – **Read bits** (per cache line, or per word to eliminate false sharing)
    • Set on *speculative loads*
    • Snooped by a committing transaction (writes by other CPU)
  – **Modified bits** (per cache line)
    • Set on *speculative stores*
    • Indicate what to *rollback* if a violation is detected
    • Different from dirty bit
  – **Renamed bits** (optional)
    • At word or byte granularity
    • To indicate local updates (WAR) that do not cause a violation
    • Subsequent reads that read lines with these bits set, they do NOT set read bits because local WAR is not considered a violation
During A Transaction Commit

• Need to collect all of the modified caches together into a commit packet

• Potential solutions
  – A separate write buffer, or
  – An address buffer maintaining a list of the line tags to be committed
  – Size?

• Broadcast all writes out as one single (large) packet to the rest of the system
Re-execute A Transaction

- Rollback is needed when a transaction cannot commit
- Checkpoints needed prior to a transaction
- Checkpoint memory
  - Use local cache
  - Overflow issue
    - Conflict or capacity misses require all the victim lines to be kept somewhere (e.g. victim cache)
- Checkpoint register state
  - Hardware approach: Flash-copying rename table / arch register file
  - Software approach: extra instruction overheads
Sample TCC Hardware

- Write buffers and L1 Transaction Control Bits
  - Write buffer in processor, before broadcast
- A broadcast bus or network to distribute commit packets
  - All processors see the commits in a single order
  - Snooping on broadcasts triggers violations, if necessary
- Commit arbitration/sequence logic
Ideal Speedups with TCC

- equake_l : long transactions
- equake_s : short transactions
Speculative Write Buffer Needs

- Only a few KB of write buffering needed
  - Set by the natural transaction sizes in applications
  - Small write buffer can capture 90% of modified state
  - Infrequent overflow can be always handled by committing early
Broadcast Bandwidth

- Broadcast is bursty
- Average bandwidth
  - Needs ~16 bytes/cycle @ 32 processors with whole modified lines
  - Needs ~8 bytes/cycle @ 32 processors with dirty data only
- High, but feasible on-chip
TCC vs MESI [PACT 2005]

- Application, Protocol + Processor count
Implementation of MIT’s LTM [HPCA 05]

- Transactional Memory should support transactions of arbitrary size and duration
- LTM — Large Transactional Memory
- No change in cache coherence protocol
- Abort when a memory conflict is detected
  - Use coherency protocol to check conflicts
  - Abort (younger) transactions during conflict resolution to guarantee forward progress
- For potential rollback
  - Checkpoint rename table and physical registers
  - Use local cache for all speculative memory operations
  - Use shared L2 (or low level memory) for non-speculative data storage
Multiple In-Flight Transactions

Original

```
XBEGIN L1
ADD R1, R1, R1
ST 1000, R1
XEND

XBEGIN L2
ADD R1, R1, R1
ST 2000, R1
XEND
```

Rename Table

```
R1 → P1, ...
```

Saved Set

```
{P1, ...}
```

• During instruction decode:
  - Maintain rename table and “saved” bits in physical registers
  - “Saved” bits track registers mentioned in current rename table
  - Constant # of set bits: every time a register is added to “saved” set we also remove one
Multiple In-Flight Transactions

<table>
<thead>
<tr>
<th>Original</th>
<th>Rename Table</th>
<th>Saved Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBEGIN L1</td>
<td>R1 → P1, ...</td>
<td>{P1, ...}</td>
</tr>
<tr>
<td>decode→ ADD R1, R1, R1</td>
<td>R1 → P2, ...</td>
<td>{P2, ...}</td>
</tr>
<tr>
<td>ST 1000, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XBEGIN L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 2000, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- When XBEGIN is decoded
  - Snapshots taken of current rename table and S bits
  - This snapshot is not active until XBEGIN retires
# Multiple In-Flight Transactions

<table>
<thead>
<tr>
<th>Original</th>
<th>Rename Table</th>
<th>Saved Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBEGIN L1</td>
<td>R1 → P1, ...</td>
<td>{P1, ...}</td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 1000, R1</td>
<td>R1 → P2, ...</td>
<td>{P2, ...}</td>
</tr>
<tr>
<td>XEND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XBEGIN L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 2000, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**decode**
## Multiple In-Flight Transactions

<table>
<thead>
<tr>
<th>Original</th>
<th>Rename Table</th>
<th>Saved Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBEGIN L1</td>
<td>R1$\rightarrow$ P1, ...</td>
<td>{P1, ...}</td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 1000, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td>R1$\rightarrow$ P2, ...</td>
<td>{P2, ...}</td>
</tr>
<tr>
<td>XBEGIN L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 2000, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multiple In-Flight Transactions

- When XBEGIN retires
  - Snapshots taken at decode become active, which will prevent P1 from reuse
  - 1st transaction queued to become active in memory
  - To abort, we just restore the active snapshot’s rename table

Original

XBEGIN L1
ADD R1, R1, R1
ST 1000, R1
XEND

decode

XBEGIN L2
ADD R1, R1, R1
ST 2000, R1
XEND

rename table

R1 → P1, ...

Saved Set

{P1, ...}

Active snapshot

retire
Multiple In-Flight Transactions

We are only reserving registers in the active set
- This implies that exactly # of arch registers are saved
- This number is strictly limited, even as we speculatively execute through multiple transactions
Multiple In-Flight Transactions

Original
XBEGIN L1
ADD R1, R1, R1
ST 1000, R1
XEND

Rename Table
R1 \rightarrow P1, ...

Saved Set
\{P1, ...\}

Saved Set
\{P2, ...\}

Saved Set
\{P3, ...\}

• Normally, P1 would be freed here
• Since it is in the active snapshot’s “saved” set, we place it onto the register reserved list
Multiple In-Flight Transactions

<table>
<thead>
<tr>
<th>Original</th>
<th>Rename Table</th>
<th>Saved Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBEGIN L1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
<td>R1 → P2, ...</td>
<td>{P2, ...}</td>
</tr>
<tr>
<td>ST 1000, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XBEGIN L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
<td>R1 → P3, ...</td>
<td>{P3, ...}</td>
</tr>
<tr>
<td>ST 2000, R1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• When XEND retires:
  - Reserved physical registers (e.g. P1) are freed, and active snapshot is cleared
  - Store queue is empty
## Multiple In-Flight Transactions

<table>
<thead>
<tr>
<th>Original</th>
<th>Rename Table</th>
<th>Saved Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBEGIN L1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 1000, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>retire XBEGIN L2</td>
<td>R1→P2, …</td>
<td>{P2, …}</td>
</tr>
<tr>
<td>ADD R1, R1, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 2000, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Second transaction becomes active in memory
Cache Overflow Mechanism

- Need to keep
  - `Current` (speculative) values
  - `Rollback` values
- Common case is commit, so keep `Current` in cache
- Problem:
  - `uncommitted current` values do not fit in local cache
- Solution
  - Overflow hashtable as extension of cache

---

### Overflow Hashtable

<table>
<thead>
<tr>
<th>key</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 1000, 55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XBEGIN L1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R1, 1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 2000, 66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 3000, 77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R1, 1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Way 0

<table>
<thead>
<tr>
<th>O</th>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Way 1

<table>
<thead>
<tr>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Legend**

- `O` = Offset
- `T` = Tag
Cache Overflow Mechanism

- **T bit per cache line**
  - Set if accessed during a transaction

- **O bit per cache set**
  - Indicate set *overflow*

- **Overflow storage in physical DRAM**
  - Allocate and resize by the OS
  - Search when miss: complexity of a page table walk
  - If a line is found, **swapped** with a line in the set

### Overflow Hashtable

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| ST 1000, 55 |
| XBEGIN L1 |
| LD R1, 1000 |
| ST 2000, 66 |
| ST 3000, 77 |
| LD R1, 1000 |
| XEND |

### Way 0

<table>
<thead>
<tr>
<th>O</th>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

### Way 1

| T | tag | data |
Cache Overflow Mechanism

- Start with non-transactional data in the cache.

```
Overflow Hashtable
key  data

ST 1000, 55

XBEGIN L1
LD R1, 1000
ST 2000, 66
ST 3000, 77
LD R1, 1000
XEND
```
### Cache Overflow Mechanism

#### Overflow Hashtable

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Way 0

<table>
<thead>
<tr>
<th>O</th>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1000</td>
<td>55</td>
</tr>
</tbody>
</table>

#### Way 1

<table>
<thead>
<tr>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Transaction leakage sets the T bit*

- **ST 1000, 55**
- **XBEGIN L1**
- **LD R1, 1000**
- **ST 2000, 66**
- **ST 3000, 77**
- **LD R1, 1000**
- **XEND**
Cache Overflow Mechanism

- Expect most transactional writes fit in the cache

Overflow Hashtable

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ST 1000, 55
XBEGIN L1
LD R1, 1000

**Way 0**

<table>
<thead>
<tr>
<th>O</th>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1000</td>
<td>55</td>
</tr>
</tbody>
</table>

**Way 1**

<table>
<thead>
<tr>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
<td>66</td>
</tr>
</tbody>
</table>

ST 2000, 66
ST 3000, 77
LD R1, 1000
XEND
**Cache Overflow Mechanism**

- A conflict miss
- Overflow sets O bit
- Replacement taken place (LRU)
- Old data spilled to DRAM (hashtable)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3000</td>
</tr>
<tr>
<td>1</td>
<td>2000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>55</td>
</tr>
</tbody>
</table>

ST 1000, 55
XBEGIN L1
LD R1, 1000
ST 2000, 66

ST 3000, 77
LD R1, 1000
XEND
## Cache Overflow Mechanism

### Miss to an overflowed line, checks overflow table
- If found, swap (like a victim cache)
- Else, proceed as miss

### Overflow Hashtable

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>77</td>
</tr>
</tbody>
</table>

### Instructions
- ST 1000, 55
- XBEGIN L1
- LD R1, 1000
- ST 2000, 66
- ST 3000, 77
- LD R1, 1000
- XEND
**Cache Overflow Mechanism**

- **Abort**
  - Invalidate all lines with T set (assume L2 or lower level memory contains original values)
  - Discard overflow hashtable
  - Clear O and T bits

- **Commit**
  - Write back hashtable; NACK interventions during this
  - Clear O and T bits in the cache

---

**Overflow Hashtable**

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>77</td>
</tr>
</tbody>
</table>

**L2**

- ST 1000, 55
- XBEGIN L1
- LD R1, 1000
- ST 2000, 66
- ST 3000, 77
- LD R1, 1000

**Way 0**

<table>
<thead>
<tr>
<th>O</th>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1000</td>
<td>55</td>
</tr>
</tbody>
</table>

**Way 1**

<table>
<thead>
<tr>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2000</td>
<td>66</td>
</tr>
</tbody>
</table>

---
LTM vs. Lock-based

![Graph comparing run time with and without locks for different benchmark applications.](image-url)
Further Readings