
Maurice Herlihy   J. Eliot B. Moss

Presented by Silvan Mosberger
Background: Transactional Memory

- Concurrency is hard! Locking is error-prone, transactional memory is easy
- Allows multiple operations, a transaction, to be executed atomically
- Can include loads/stores to arbitrary memory locations
- Transactions are isolated, all its changes are only visible once it commits
- When something went wrong, abort it and retry
Problems

- Problems with conventional locking techniques in highly concurrent systems
  - Priority Inversion
  - Lock convoy
  - Deadlock
- Software transactional memory is nice but slow

Goal

- Specify implementation for hardware transactional memory
- Make it fast in highly concurrent systems
- Consequently, committing/aborting transactions should be processor-local
Key Approach and Idea

Idea

– Snoopy cache coherency protocol can also detect conflicting transactions
– Abort a transaction upon conflict

Key Approach

– Additional smaller transactional cache for memory locations participating in the transaction
– Use two cache entries, one in case of abort, one in case of commit
– Extend snoopy protocol for transactions
Mechanisms: Programmer Interface

- **LT**: Load-transactional, read a memory location
- **LTX**: Load-transactional-exclusive, read a memory location “hinting” it will be updated
- **ST**: Store-transactional, write a memory location
- **COMMIT**: attempt to commit the changes
- **ABORT**: discard all changes
- **VALIDATE**: Test for already aborted, guarantees consistency of previously read values
Mechanisms: Cache structure

Main Memory

Bus

1. Normal cache
   Trans. cache

2. Normal cache
   Trans. cache

3. Normal cache
   Trans. cache
Mechanisms: Transactional cache

<table>
<thead>
<tr>
<th>Name</th>
<th>Access</th>
<th>Shared?</th>
<th>Modified?</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID</td>
<td>none</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>VALID</td>
<td>R</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DIRTY</td>
<td>R, W</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>RESERVED</td>
<td>R, W</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1: Cache line states

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTY</td>
<td>contains no data</td>
</tr>
<tr>
<td>NORMAL</td>
<td>contains committed data</td>
</tr>
<tr>
<td>XCOMMIT</td>
<td>discard on commit</td>
</tr>
<tr>
<td>XABORT</td>
<td>discard on abort</td>
</tr>
</tbody>
</table>

Table 2: Transactional tags
# Mechanisms: Bus cycles

<table>
<thead>
<tr>
<th>Name</th>
<th>Kind</th>
<th>Meaning</th>
<th>New access</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>regular</td>
<td>read value</td>
<td>shared</td>
</tr>
<tr>
<td>RFO</td>
<td>regular</td>
<td>read value</td>
<td>exclusive</td>
</tr>
<tr>
<td>WRITE</td>
<td>both</td>
<td>write back</td>
<td>exclusive</td>
</tr>
<tr>
<td>T_READ</td>
<td>trans</td>
<td>read value</td>
<td>shared</td>
</tr>
<tr>
<td>T_RFO</td>
<td>trans</td>
<td>read value</td>
<td>exclusive</td>
</tr>
<tr>
<td>BUSY</td>
<td>trans</td>
<td>refuse access</td>
<td>unchanged</td>
</tr>
</tbody>
</table>

## Standard bus cycles
- **WRITE**: Write back to main memory
- **READ**: Read for shared access
- **RFO**: Read for exclusive access

## New transactional cycles
- **T_READ**: Same as READ but for transactional cache
- **T_RFO**: Same as RFO but for transactional cache
- **BUSY**: Used for refusing cache requests
Mechanisms: Processor Actions

```cpp
// Whether a transaction is in progress
bool TACTIVE;
// Whether the transaction is still active or aborted
bool TSTATUS;

void abort(bool internal) {
    if (internal) {
        TSTATUS = false;
    } else {
        TACTIVE = false;
    }
    set_all(XCOMMIT, NORMAL);
    set_all(XABORT, EMPTY);
}

bool commit() {
    bool oldStatus = TSTATUS;
    TACTIVE = false;
    TSTATUS = true;
    set_all(XCOMMIT, EMPTY);
    set_all(XABORT, NORMAL);
    return oldStatus;
}

bool validate() {
    bool oldStatus = TSTATUS;
    if (!TSTATUS) {
        TACTIVE = false;
        TSTATUS = true;
    }
    return oldStatus;
}
```
Mechanisms: Processor Action: LT

- Cache lookup result
  - LT
  - XABORT <data>
  - LT
  - NORMAL <data>
  - LT
  - (no entry)

Turns into

- Success <data>
- XABORT <data>
- XCOMMIT <data>

Create entries

Main Memory

Abort transaction

SUCCESS

T_READ
Mechanisms: Processor Action: LTX

- LTX
- XABORT <data>
- NORMAL <data>
- XABORT <data>
- XCOMMIT <data>
- T_RFO
- SUCCESS <data>
- BUSY
- Create entries with RESERVED
- Abort transaction

Cache lookup result

Main Memory
Mechanisms: Processor Action: ST

- **ST <new data>** → XABORT <data>
- **ST <new data>** → NORMAL <data>
- **ST <new data>** → (no entry)

- **Cache lookup result**:
  - XABORT <data> → XABORT <new data>
  - NORMAL <data> → XABORT <new data>
  - (no entry) → T_RFO

- **Turns into**:
  - XABORT <new data> → XCOMMIT <data>
  - XABORT <new data> → (no entry)

- **Create entries with RESERVED**:
  - XCOMMIT <data> → Success <data>

- **Main Memory**:
  - BUSY

- **Abort transaction**
Key Results: Methodology and Evaluation

Architectures

- **Bus:** Snoopy cache coherence for bus-based architecture
- **Network:** Chaiken directory protocol for network-based machine, discussed in technical report

Benchmarks:

- **Counting:** Increment shared counter. Short critical sections → contention high
- **Producer/Consumer:** Shared bounded FIFO buffer, half of the processors producers, half consumers
- **Doubly-Linked List:** Shared linked list, every process dequeues from tail, enqueues back to head. No easy concurrency for locks
Key Results: Methodology and Evaluation

Other techniques for comparison:

- **TTS (test-and-test-and-set) Lock**: Read cached value until evicted, then do test-and-set in memory directly

- **LL/SC (load-linked/store-cond)**: LL copies value to local variable, SC tries to change its value and succeeds if no other process has modified it

- **MCS Lock (software queueing)**: Placed on queue if unable to acquire lock, eliminating lock polls

- **QOSB (hardware queueing)**: Queue incorporated into cache coherence protocol via unused cache lines
Benchmark: Counting
Benchmark: Producer/Consumer

Bus

Network

Elapsed Time (in cycles x 1000)

0
10
20
30 Concurrency

MCS Lock
LL/SC Lock
TTS Lock
QOSB

Trans. Mem.

MCS Lock
LL/SC Lock
TTS Lock
QOSB

Trans. Mem.
Benchmark: Doubly-Linked List

Elapsed Time (in cycles × 1000)

- TTS Lock
- LL/SC Lock
- MCS Lock
- QOSB
- Trans. Mem.

Concurrency

Bus

Elapsed Time (in cycles × 1000)

- MCS Lock
- LL/SC Lock
- TTS Lock
- QOSB
- Trans. Mem.

Concurrency

Network
Summary

- **Problem:** Locks are fast but hard to use, software transactional memory is easy to use but slow
- **Goal:** Implement fast hardware-based transactional memory
- **Idea:** Use separate transactional cache for storing two entries for every memory location, one in case of commit, one in case of abort. Use cache coherency protocol to detect conflicting transactions.
- **Results:** Hardware transactional memory outperforms other techniques especially in highly concurrent systems.
Strengths

- Explains the limits of this approach and how to work around it
  - Starvation $\rightarrow$ exponential backoff
  - Too few cache lines $\rightarrow$ emulate in software
- First paper to fully explore hardware transactional memory
- No need to write back to memory on commit, happens over time when cache lines get replaced
- Extra technical paper explains everything in much more detail
Weaknesses

- No explanation as to why the mentioned protocol is correct or how it came to be
- No diagrams to visualize the protocol, not easy to follow with just text
- XABORT means “Discard on abort”, but it becomes valid on commit which is more understandable. Same with XCOMMIT. Naming is hard!
- LTX and XCOMMIT only there to make it faster, but no benchmarks for determining the difference they make and in which cases
- Doesn’t explain well how transactional and non-transactional memory locations interact
Takeaways

- Consider using transactional memory for your concurrency needs
- Ideas sometimes have applications you haven’t thought of initially
- Consider tradeoffs, it might be desirable to have more complexity for more performance
Questions and Open Discussion
Extra Questions

- Has anybody used transactional memory before?
- How could software transactional memory be implemented?
  - Write to shared memory
  - Log all read and writes
  - On commit, ensure all reads haven’t changed
  - Abort and roll back changes if not
- What problem is there with not writing immediately back to main memory?
  - Values might not get written back for a while → more chance of losing updates on power loss
  - Pollutes cache, creating new entries can require writeback to main memory
Related Papers


Extra: Usage example

```
shared long[] accounts = ...;

// Try to transfer `amount` of money from account `from` to `to`
int transfer (int from, int to, long amount) {
    float frombalance = LTX(&accounts[from]);
    if (amount >= frombalance)
        ABORT();

    float tobalance = LTX(&accounts[to]);
    ST(&accounts[from], frombalance - amount);
    ST(&accounts[to], tobalance + amount);
    return COMMIT();
}
```
Extra: Snoopy actions

- Both caches snoop on the bus
- A cache ignores any cycles for lines not in that cache
- The regular cache
  - On READ/T_READ, if state is VALID, return value
  - If RESERVED or DIRTY, return value and reset to VALID
  - On RFO/T_RFO, return data and invalidate line
- The transactional cache
  - If TSTATUS is false, or if READ/RFO, behave just like normal cache, except it ignores entries with transactional tag != NORMAL
  - On T_READ and state VALID, return value
  - Otherwise return BUSY
- Either cache can do WRITE when line needs to be replaced
- Memory only responds to READ, T_READ, RFO and T_RFO that no cache responds to, and all WRITE requests