Unbounded Transactional Memory

C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, Sean Lie

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{cananian, krste, bradley, cel}@mit.edu, sean@slie.ca

Thanks to Marty Deneroff (then at SGI)

This research supported in part by a DARPA HPCS grant with SGI, DARPA/AFRL Contract F33615-00-C-1692, NSF Grants ACI-0324974 and CNS-0305606, NSF Career Grant CCR00093354, and the Singapore-MIT Alliance
**Transactional Memory (definition)**

- A transaction is a sequence of memory loads and stores that either commits or aborts.
- If a transaction commits, all the loads and stores appear to have executed atomically.
- If a transaction aborts, none of its stores take effect.
- Transaction operations aren't visible until they commit or abort.
- Simplified version of traditional ACID database transactions (no durability, for example).
- For this talk, we assume no I/O within transactions.
void pushFlow(Vertex v1, Vertex v2, double flow) {
    lock_t lock1, lock2;
    if (v1.id < v2.id) { /* avoid deadlock */
        lock1 = v1.lock; lock2 = v2.lock;
    } else {
        lock1 = v2.lock; lock2 = v1.lock;
    }
    lock(lock1);
    lock(lock2);
    if (v2.excess > f) {
        /* move excess flow */
        v1.excess += f;
        v2.excess -= f;
    }
    unlock(lock2);
    unlock(lock1);
}
Invisible transactions?

• Rajwar & Goodman: Speculative Lock Elision and Transactional Lock Removal
  - speculatively identify locks; make xactions

• Martinez & Torrellas: Speculative Synchronization
  - guarantee fwd progress w/ non-speculative thread

Keep transactions visible
Infrequent, Small, Mostly-Serial?

To date, xactions assumed to be:

- **Small**
  - BBN Pluribus (~1975): 16 clock-cycle bus-locked “transaction”
  - Knight; Herlihy & Moss: transactions which fit in cache
- **Infrequent**
  - Software Transactional Memory (Shavit & Touitou; Harris & Fraser; Herlihy et al)
- **Mostly-serial**
  - Transactional Coherence & Consistency (Hammond, Wong, et al)
Transact-ifying Linux

- Experiment to discover xaction properties of large real-world app.
  - First complete OS investigated!
- **User-Mode Linux 2.4.19**
  - instrumented every load and store, all locks
  - locks\(\rightarrow\)xactions; locks not held over I/O!
  - run 2-way SMP (two processes; two processors)
- **Two workloads**
  - Parallel make of Linux kernel ('make linux')
  - dbench running three clients
- Run program to get a trace; run trace through **Transactional Memory simulator**
  - 1MB 4-way set-associative 64-byte-line cache
  - Paper also has simulation runs for SpecJVM98
**TM Cache-size requirements (Linux)**

- # of overflowing xactions as a function of (fully-associative) cache size for `make_linux` & `dbench`
- Almost all of the transactions require < 100 cache lines
  - 99.9% need fewer than 54 cache lines
- There are, however, some very large transactions!
  - >500k-byte fully-associative cache required
May Be Large, Frequent, and Concurrent

- Lots of small xactions
  - Millions of xactions in these benchmarks
  - Problem for software-only schemes

- Significant tail: large xactions are few, but very large
  - Thousands of cache lines touched
  - Problem for bounded transactional schemes

- Potential for additional concurrency
  - Distribution of hot cache lines suggest that 4× more concurrency may be possible on our Linux benchmarks

Programmers want unbounded xactions…
Transactional Programming

- Locks: the devil we know
- Complex sync techniques: library-only
  - Nonblocking synchronization
  - Bounded transactions
    - Compilers don't expose memory references (Indirect dispatch, optimizations, constants)
    - Not portable! Changing cache-size breaks apps.

- Unbounded Transactions:
  - Can be thought about at high-level
  - Match programmer's intuition about atomicity
  - Allow black box code to be composed safely
  - Promise future excitement!
    - Fault-tolerance / exception-handling
    - Speculation / search
LTM: Visible, Large, Frequent, Scalable

- "Large Transactional Memory"
  - not truly unbounded, but simple and cheap
- Minimal architectural changes, high performance
  - Small mods to cache and processor core
  - No changes to main memory, cache coherence protocols or messages
  - Can be pin-compatible with conventional proc
- Design presented here based on SGI Origin 3000 shared-memory multi-proc
  - distributed memory
  - directory-based write-invalidate coherency protocol
Two new instructions

• **XBEGIN** \( pc \)
  - Begin a new transaction. Entry point to an *abort handler* specified by \( pc \).
  - If transaction must fail, roll back processor and memory state to what it was when **XBEGIN** was executed, and jump to \( pc \).
    - Think of this as a mispredicted branch.

• **XEND**
  - End the current transaction. If **XEND** completes, the xaction is committed and appeared atomic.

• Nested transactions are subsumed into outer transaction.
**Transaction Semantics**

- **Two transactions**
  - “A” has an abort handler at L1
  - “B” has an abort handler at L2
    - Here, very simplistic retry. Other choices!

- **Always need “current” and “rollback” values for both registers and memory**
Handling conflicts

We need to track locations read/written by transactional and non-transactional code.

When we find a conflict, transaction(s) must be aborted.
- We always “kill the other guy”
- This leads to non-blocking systems.

**Processor 1**
- XBEGIN L1
- ADD R1, R1, R1
- ST 1000, R1
- XEND

**L2: Processor 2**
- XBEGIN L2
- ADD R1, R1, R1
- ST 2000, R1
- XEND
- ST 1000, 65
Restoring register state

- Minimally invasive changes; build on existing rename mechanism
- Both “current” and “rollback” architectural register values stored in physical registers
- In conventional speculation, “rollback” values stored until the speculative instruction graduates (order 100 instrs)
- Here, we keep these until the transaction commits or aborts (unbounded # of instrs)
- But we only need one copy!
  - only one transaction in the memory system per processor
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

- This example has two transactions, with abort handlers at L1 and L2
- Assume instruction window of length 5
  - allows us to speculate into next transaction(s)
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></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>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>

- During instruction decode:
  - Maintain rename table and "saved" bits
  - "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

### Original
- XBEGIN L1
- ADD P2, P1, P1
- ST 1000, R1
- XEND
- XBEGIN L2
- ADD R1, R1, R1
- ST 2000, R1
- XEND

### Rename Table
- R1 → P1, ...
- R1 → P2, ...

### Saved set
- { P1, ... }
- { P2, ... }

- **When XBEGIN is decoded:**
  - Snapshots taken of current Rename table and S-bits.
  - This snapshot is not active until XBEGIN graduates.
### 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 P2, P1, P1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 1000, P2</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>
Multiple in-flight transactions

**Original**

- XBEGIN L1
- ADD P2, P1, P1
- ST 1000, P2
- XEND
- XBEGIN L2
- ADD R1, R1, R1
- ST 2000, R1
- XEND

**Rename Table**

- R1 → P1, ...

**Saved set**

- \{ P1, ... \}

- R1 → P2, ...

- \{ P2, ... \}
Multiple in-flight transactions

- When XBEGIN graduates:
  - Snapshot taken at decode becomes active, which will prevent P1 from being reused
  - 1st transaction queued to become active in memory
  - To abort, we just restore the active snapshot's rename table

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

- We're only reserving registers in the active set
  - This implies that exactly #AR registers are saved
  - This number is strictly limited, even as we speculatively execute through multiple xactions
Multiple in-flight transactions

- Normally, P1 would be freed here
- Since it's in the active snapshot's "saved" set, we put it on the register reserved list instead
**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→P2, ...</td>
<td>{ P2, ... }</td>
</tr>
<tr>
<td>ADD P2, P1, P1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 1000, P2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XEND</td>
<td>R1→P3, ...</td>
<td>{ P3, ... }</td>
</tr>
<tr>
<td>XBEGIN L2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD P3, P2, P2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 2000, P3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **When XEND graduates:**
  - Reserved physical registers (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>R1→P2, ...</td>
<td>{ P2, ... }</td>
</tr>
<tr>
<td>ADD P2, P1, P1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 1000, P2</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 P3, P2, P2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST 2000, P3</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” values as well as “rollback” values
  - Common-case is commit, so keep “current” in cache
  - What if uncommitted “current” values don't all fit in cache?
- Use overflow hashtable as extension of cache
  - Avoid looking here if we can!

<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>
</tbody>
</table>

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>O</strong></td>
<td><strong>T</strong></td>
<td><strong>tag</strong></td>
<td><strong>data</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Way 0</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Way 1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **T** bit per cache line
  - set if accessed during xaction
- **O** bit per cache set
  - indicates set overflow
- **Overflow storage in physical DRAM**
  - allocated/resized by OS
  - probe/miss: complexity of search ≈ page table walk

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

- Start with non-transactional data in the cache

<table>
<thead>
<tr>
<th></th>
<th>Way 0</th>
<th></th>
<th>Way 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>55</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Overflow hashtable

- key
- data

ST 1000, 55
XBEGIN L1
LD R1, 1000
ST 2000, 66
ST 3000, 77
LD R1, 1000
XEND
Cache overflow: recording reads

- Transactional read sets the T bit.

**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>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>

Overflow hashtable

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

**Commands:**
- ST 1000, 55
- XBEGIN L1
- LD R1, 1000
- ST 2000, 66
- ST 3000, 77
- LD R1, 1000
- XEND
### Cache overflow: recording writes

#### Most transactional writes fit in the cache.

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

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

**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
Cache overflow: spilling

Overflow sets O bit
New data replaces LRU
Old data spilled to DRAM

Overflow hashtable

<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

Way 0

<table>
<thead>
<tr>
<th>O</th>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>T</td>
<td>3000</td>
<td>77</td>
</tr>
</tbody>
</table>

Way 1

<table>
<thead>
<tr>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>2000</td>
<td>66</td>
</tr>
</tbody>
</table>
Cache overflow: miss handling

- Miss to an overflowed line checks overflow table
- If found, swap overflow and cache line; proceed as hit
- Else, proceed as miss.

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

Overflow hashtable

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

```
ST 1000, 55
XBEGIN L1
LD R1, 1000
ST 2000, 66
ST 3000, 77
LD R1, 1000
XEND
```
### Cache overflow: commit/abort

#### Abort:
- Invalidate all lines with T set
- Discard overflow hashtable
- Clear O and T bits

#### Commit:
- Write back hashtable; NACK interventions during this
- Clear O and T bits

---

**Overflow hashtable**

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

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

---

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

<table>
<thead>
<tr>
<th>T</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>2000</td>
<td>66</td>
</tr>
</tbody>
</table>
Cycle-level LTM simulation

- LTM implemented on top of UVSIM (itself built on RSIM)
  - shared-memory multiprocessor model
  - directory-based write-invalidate coherence
- Contention behavior:
  - C microbenchmarks w/ inline assembly
  - Up to 32 processors
- Overhead measurements:
  - Modified MIT FLEX Java compiler
  - Compared no-sync, spin-lock, and LTM xaction
  - Single-threaded, single processor

RSIM  Flex  Compiler Infrastructure  GCC  Java
**Contention behavior**

![Graph showing contention behavior](image)

- **Contention microbenchmark: 'Counter'**
  - 1 shared variable; each processor repeatedly adds
  - locking version uses global LLSC spinlock
  - **Small xactions commit even with high contention**
  - Spin-lock causes lots of cache interventions even when it can't be taken (standard SGI library impl)
LTM Overhead: SPECjvm98

Run time, % of no-sync time

- With Locks
- Other
- In Xaction
- Overflow

Benchmark application

(check, compress, jess, db, javac, mpeg-audio, jack)

100%

0%

Ananian/Asanović/Kuszmaul/Leiserson/Lie: Unbounded Transactional Memory, HPCA '05
Is this good enough?

- Problems solved:
  - Xactions **as large as physical memory**
  - **Scalable** overflow and commit
  - Easy to **implement**!
  - Low overhead
  - May **speed up** Linux!

- Open Problems...
  - Is “physical memory” **large enough**?
  - What about **duration**?
  - **Time-slice interrupts**!
Beyond LTM: UTM

- We can do better!
- The UTM architecture allows transactions as large as virtual memory, of unlimited duration, which can migrate without restart
- Same `xbegin pc/xend` ISA; same register rollback mechanism
- Canonical transaction info is now stored in single `xstate` data struct in main memory
xstate data structure

**Application Memory**

- **Commit Log Entry**
- **Transaction Log**
  - Log Entry
    - Rollback values
    - Blk Ptr
    - Next Reader

- **Commit record**:
  - **PENDING**, **COMMITTED**, **ABORTED**
- **vector of log entries** w/ “rollback” values
  - each corresponds to a block in main memory

- **Log ptr & RW bit** for each application memory block
  - **Log ptr/next reader** form linked list of all log entries for a given block

---

(38) Ananian/Asanović/Kuszmaul/Leiserson/Lie: Unbounded Transactional Memory, HPCA ’05
Caching in UTM

- Most log entries don't need to be created
- Transaction state stored in cache/overflow DRAM and monitored using cache-coherence, as in LTM
- Only create transaction log when thread is descheduled, or run out of physical mem.
- Can discard all log entries when xaction commits or aborts
  - Commit – block left in X state in cache
  - Abort – use old value in main memory
- In-cache representation need not match xstate representation
Performance/Limits of UTM

• Limits:
  – More-complicated implementation
    • Best way to create xstate from LTM state?
  – Performance impact of swapping.
    • When should we abort rather than swap?

• Benefits:
  – Unlimited footprint
  – Unlimited duration
  – Migration and paging possible
  – Performance may be as fast as LTM in the common case
Conclusions

• First look at **xaction properties of Linux:**
  - 99.9% of xactions touch \( \leq 54 \) cache lines
  - but may touch \( > 8000 \) cache lines
  - 4x concurrency?

• **Unbounded, scalable, and efficient**
  Transactional Memory systems can be built.
  - Support large, frequent, and concurrent xactions
  - What *could* software for these look like?
    • Allow programmers to (finally!) use our parallel systems!

• **Two implementable architectures:**
  - LTM: easy to realize, almost unbounded
  - UTM: truly unbounded
Open questions

- I/O interface?
- Transaction ordering?
  - Sequential threads provide inherent ordering
- Programming model?
- Conflict resolution strategies