Transactional Memory

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Our Vision for the Future

In this course, we covered ....
Best practices ...
New and clever ideas ...
And common-sense observations.
Our Vision for the Future

In this course, we covered ....

Nevertheless …

Concurrent programming is still too hard …

Here we explore why this is …. 

And what we can do about it.
Locking
Coarse-Grained Locking

Easily made correct …
But not scalable.
Fine-Grained Locking

Can be tricky …
Locks are not Robust

If a thread holding a lock is delayed ...
Locking Relies on Conventions

• Relation between
  – Lock bit and object bits
  – Exists only in programmer’s

/*
 * When a locked buffer is visible to the I/O layer
 * BH_Launder is set. This means before unlocking
 * we must clear BH_Launder,mb() on alpha and then
 * clear BH_Lock, so no reader can see BH_Launder set
 * on an unlocked buffer and then risk to deadlock.
 */

Actual comment from Linux Kernel
(hat tip: Bradley Kuszmaul)
Simple Problems are hard

enq(x)  double-ended queue  enq(y)

No interference if ends “far apart”

Interference OK if queue is small

Clean solution is publishable result:
[Michael & Scott PODC 97]
Locks Not Composable

Transfer item from one queue to another

Must be atomic:
No duplicate or missing items
Locks Not Composable

Lock source
Unlock source & target
Lock target
Locks Not Composable

Methods cannot provide internal synchronization

Objects must expose locking protocols to clients

Clients must devise and follow protocols

Abstraction broken!
Monitor Wait and Signal

If buffer is empty, wait for item to show up.
Wait and Signal do not Compose

Wait for either?
The Transactional Manifesto

- Current practice inadequate
  - to meet the multicore challenge
- Research Agenda
  - Replace locking with a transactional API
  - Design languages or libraries
  - Implement efficient run-times
Transactions

Block of code ....

Atomic: appears to happen instantaneously

Serializable: all appear to happen in one-at-a-time order

Commit: takes effect (atomically)

Abort: has no effect (typically restarted)
Atomic Blocks

```java
atomic {
    x.remove(3);
    y.add(3);
}

atomic {
    y = null;
}
```
Atomic Blocks

```java
atomic {
    x.remove(3);
    y.add(3);
}

atomic {
    y = null;
}
```

No data race
A Double-Ended Queue

```java
public void LeftEnq(item x) {
    Qnode q = new Qnode(x);
    q.left = left;
    left.right = q;
    left = q;
}
```

Write sequential Code
public void LeftEnq(item x) {
    Qnode q = new Qnode(x);
    q.left = left;
    left.right = q;
    left = q;
}
A Double-Ended Queue

```
public void LeftEnq(item x) {
    atomic {
        Qnode q = new Qnode(x);
        q.left = left;
        left.right = q;
        left = q;
    }
}
```

Enclose in atomic block
Warning

- Not always this simple
  - Conditional waits
  - Enhanced concurrency
  - Complex patterns
- But often it is…
Composition?
Composition?

```java
public void Transfer(Queue<T> q1, q2) {
    atomic {
        T x = q1.deq();
        q2.enq(x);
    }
}

Trivial or what?
```
public T LeftDeq() {
    atomic {
        if (left == null) {
            retry;
        }
        ...
    }
}
Composable Conditional Waiting

```java
atomic {
  x = q1.deq();
} orElse {
  x = q2.deq();
}
```

Run 1st method. If it retries …
Run 2nd method. If it retries …
Entire statement retries
Hardware Transactional Memory

- Exploit Cache coherence
- Already almost does it
  - Invalidation
  - Consistency checking
- Speculative execution
  - Branch prediction = optimistic synch!
HW Transactional Memory

read

active

T

Interconnect

memory

caches
Transactional Memory

Active

Read

Caches

Memory

Art of Multiprocessor Programming
Transactional Memory

committed

active

T

T

caches

memory
Transactional Memory

- Committed
- Active
- Memory write
- Caches
- Memory
Rewind

aborted

write

active

caches

memory
Transaction Commit

• At commit point
  – If no cache conflicts, we win.

• Mark transactional entries
  – Read-only: valid
  – Modified: dirty (eventually written back)

• That’s all, folks!
  – Except for a few details …
Hardware Transactional Memory (HTM)

IBM’s Blue Gene/Q & System Z & Power8

Intel’s Haswell TSX extensions
if (_xbegin() == _XBEGIN_STARTED) {
    speculative code
    _xend()
} else {
    abort handler
}
Intel RTM

if (_xbegin() == XBEGIN_STARTED) {
    speculative code
    _xend()
} else {
    abort handler
}

start a speculative transaction
Intel RTM

if (_xbegin() == _XBEGIN_STARTED) {
    speculative code
    _xend()
} else {
    abort handler
}

If you see this, you are inside a transaction
if (_xbegin() == _XBEGIN_STARTED) {
    speculative code
    _xend()
} else {
    abort handler
}
if (_xbegin() == _XBEGIN_STARTED) {
    speculative code
    _xend()
} else {
    abort handler
}

you could retry the transaction, or take an alternative path
Abort codes

```c
if (_xbegin() == _XBEGIN_STARTED) {
    speculative code
} else if (status & _XABORT_EXPLICIT) {
    aborted by user code
} else if (status & _XABORT_CONFLICT) {
    read-write conflict
} else if (status & _XABORT_CAPACITY) {
    cache overflow
} else {
    ...
}
```
Abort codes

```c
if (_xbegin() == _XBEGIN_STARTED) {
    speculative code
} else if (status & _XABORT_EXPLICIT) {
    aborted by user code
} else if (status & _XABORT_CONFLICT) {
    read-write conflict
} else if (status & _XABORT_CAPACITY) {
    cache overflow
} else {
    ...
}
```
Abort codes

if (_xbegin() == _XBEGIN_STARTED) {
speculative code
}
else if (status & _XABORT_EXPLICIT) {
aborted by user code
}
else if (status & _XABORT_CONFLICT) {
read-write conflict
}
else if (status & _XABORT_CAPACITY) {
cache overflow
}
else {
...
}
Abort codes

if (_xbegin() == _XBEGIN_STARTED) {
    speculative code
} else if (status & _XABORT_EXPLICIT) {
    aborted by user code
} else if (status & _XABORT_CONFLICT) {
    read-write conflict
} else if (status & _XABORT_CAPACITY) {
    cache overflow
} else {
    ...
}
Abort codes

if (_xbegin() == _XBEGIN_STARTED) {
    speculative code
} else if (status & _XABORT_EXPLICIT) {
    aborted by user code
} else if (status & _XABORT_CONFLICT) {
    read-write conflict
} else if (status & _XABORT_CAPACITY) {
    cache overflow
} else {
    ...
}
RTM

Small & Medium Transactions
Best-effort
Conflicts
Overflow
Unsupported Instructions
Interrupts

Needs software fallback
Non-Speculative Fallback

```c
if (_xbegin() == _XBEGIN_STARTED) {
    read lock state
    if (lock taken) _xabort();
    work;
    _xend()
} else {
    lock->lock();
    work;
    lock->unlock();
}
```
if (_xbegin() == XBEGIN_STARTED) {
    read lock state
    if (lock taken) _xabort();
    work;
    _xend()
} else {
    lock-
    lock();
    work;
    lock-
    unlock();
}

Non-Speculative Fallback

reading lock ensures that transaction will abort if another thread acquires lock
if (_xbegin() == XBEGIN_STARTED) {
    read lock state
    if (lock taken) _xabort();
    work;
    _xend()
} else {
    lock();
    work;
    lock();
}
Non-Speculative Fallback

on abort, acquire lock & do work (aborting concurrent speculative transactions)
Lock Elision

- `<HLE_Aquire_Prefix> Lock(L)

  Atomic region executed as a \textit{transaction} or \textit{mutually exclusive on L}

- `<HLE_Release_Prefix> Release(L)

  Execute optimistically, without any locks

  Track Read and Write Sets

  Abort on memory conflict: rollback acquire lock
Lock Elision

<HLE acquire prefix> lock();
do work;
<HLE release prefix> unlock()
Lock Elision

```c
<HLE acquire prefix> lock();
do work;
<HLE release prefix> unlock()
```

first time around, read lock and execute speculatively
Lock Elision

```
<HLE acquire prefix>  lock();
do work;
<HLE release prefix>  unlock()
```

if speculation fails, no more Mr. Nice Guy, acquire the lock
Conventional Locks

lock transfer latencies

serialized execution

locks
Lock Elision

locks

lock elision
Lock Teleportation
Lock Teleportation

read transaction
Lock Teleportation

read transaction
Lock Teleportation

no locks acquired
Not all Skittles and beer

• Limits to
  – Transactional cache size
  – Scheduling quantum

• Transaction cannot commit if it is
  – Too big
  – Too slow
  – Actual limits platform-dependent
HTM Strengths & Weaknesses

• Ideal for lock-free data structures
HTM Strengths & Weaknesses

• Ideal for lock-free data structures

• Practical proposals have limits on
  – Transaction size and length
  – Bounded HW resources
  – Guarantees vs best-effort
HTM Strengths & Weaknesses

• Ideal for lock-free data structures
• Practical proposals have limits on
  – Transaction size and length
  – Bounded HW resources
  – Guarantees vs best-effort

• On fail
  – Diagnostics essential
  – Retry in software?
Locks don’t compose, transactions do.

Composition necessary for Software Engineering.

But practical HTM doesn’t really support composition!

Why we need STM
Transactional Consistency

- Memory Transactions are collections of reads and writes executed atomically.
- They should maintain consistency:
  - *External*: with respect to the interleavings of other transactions (*linearizability*).
  - *Internal*: the transaction itself should operate on a consistent state.
A Simple Lock-Based STM

• STMs come in different forms
  – Lock-based
  – Lock-free
• Here: a simple lock-based STM
• Let's start by Guaranteeing External Consistency
Synchronization

• Transaction keeps
  – Read set: locations & values read
  – Write set: locations & values to be written

• Deferred update
  – Changes installed at commit

• Lazy conflict detection
  – Conflicts detected at commit
STM: Transactional Locking

Application Memory

Map

Array of version #s & locks
## Encounter Order Locking (Undo Log)

<table>
<thead>
<tr>
<th>Undo</th>
<th>Mem</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Read
- Check unlocked, add to read set

### Write
- Lock, add value to write set

### Validate
- Check read version #s
- Increment write version #s
- Unlock write set
Commit Time Locking (Write Buff)

<table>
<thead>
<tr>
<th>Mem</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>V# 0</td>
</tr>
<tr>
<td>X</td>
<td>V#+1 0</td>
</tr>
<tr>
<td>Y</td>
<td>V# 0</td>
</tr>
<tr>
<td></td>
<td>V#+1 0</td>
</tr>
<tr>
<td></td>
<td>V# 0</td>
</tr>
<tr>
<td></td>
<td>V# 0</td>
</tr>
</tbody>
</table>

Read
- In write set? unlocked?
- Add value to read set

Write
- Add value to write set,

Validate
- Acquire locks
- Check version #s unchanged
- Install changes,
- Increment version #s, unlock
Commit Time Locking (Write Buff)

1. To Read: load lock + location
2. Location in write-set? (Bloom Filter)
3. Check unlocked add to Read-Set
4. To Write: add value to write set
5. Acquire Locks
6. Validate read/write v#’s unchanged
7. Release each lock with v#+1

Hold locks for very short duration
COM vs. ENC High Load

Red-Black Tree 20% Delete 20% Update 60% Lookup

ops/sec

threads

Hand
COM
ENC
MCS
COM vs. ENC Low Load

Red-Black Tree 5% Delete 5% Update 90% Lookup
Problem: Internal Inconsistency

- A Zombie is an active transaction destined to abort.
- If Zombies see inconsistent states bad things can happen
Internal Consistency

Invariant: \( x = 2y \)

Transaction A: reads \( x = 4 \)

Transaction B: writes 8 to \( x \), 16 to \( y \), aborts A

Transaction A: (zombie)
reads \( y = 4 \)
computes \( \frac{1}{(x-y)} \)

Divide by zero FAIL!
Solution: The Global Clock (The TL2 Algorithm)

- Have one shared global clock
- Incremented by (small subset of) writing transactions
- Read by all transactions
- Used to validate that state worked on is always consistent
Read-Only Transactions

Mem

Locks

- 12
- 32
- 56
- 19
- 17

100

- Shared Version Clock
- Private Read Version (RV)

Copy version clock to local read version clock
Read-Only Transactions

Copy version clock to local read version clock.

Read lock, version #, and memory

Shared Version Clock

Private Read Version (RV)
Read-Only Transactions

Mem

Locks

Copy version clock to local

Read lock, version #, and memory

On Commit:
check unlocked & version #s less than local read clock

12

19

17

Shared Version Clock

Private Read Version (RV)

100

100

100
Read-Only Transactions

We have taken a snapshot without keeping an explicit read set!
Example Execution: Read Only Trans

1. RV ← Shared Version Clock
2. On Read: read lock, read mem, read lock: check unlocked, unchanged, and v# ≤ RV
3. Commit.

Reads form a snapshot of memory. No read set!
Regular (Writing) Transactions

Mem

<table>
<thead>
<tr>
<th>Locks</th>
<th>12</th>
<th>32</th>
<th>56</th>
<th>19</th>
<th>17</th>
</tr>
</thead>
</table>

Copy version clock to local read version clock

Shared Version Clock

Private Read Version (RV)
Regular Transactions

Copy version clock to local read version clock.

On read/write, check:
1. Unlocked & version # < RV
2. Add to R/W set

Shared Version Clock
Private Read Version (RV)
On Commit

Acquire write locks

Shared Version Clock

Private Read Version (RV)
On Commit

Mem

Locks

Acquire write locks
Increment Version Clock

Shared Version Clock

Private Read Version (RV)
On Commit

Acquire write locks
Increment Version Clock
Check version numbers ≤ RV

Shared Version Clock
Shared Version Clock
Private Read Version (RV)
Private Read Version (RV)
On Commit

Acquire write locks
Increment Version Clock
Check version numbers < RV
Update memory

Shared Version Clock
Private Read Version (RV)
On Commit

Acquire write locks
Increment Version Clock
Check version numbers < RV
Update memory
Update write version #s
Example: Writing Trans

1. \( RV \leftarrow \text{Shared Version Clock} \)
2. On Read/Write: check unlocked and \( v\# \leq RV \) then add to Read/Write-Set
3. Acquire Locks
4. \( WV = F&I(VClock) \)
5. Validate each \( v\# \leq RV \)
6. Release locks with \( v\# \leftarrow WV \)

Reads+Inc+Writes = serializable
Version Clock Implementation

- Notice: Version clock rate is a progress concern, not a safety concern...
  - (GV4) if failed clock CAS use Version Clock set by winner.
  - (GV5) WV = Version Clock + 2; inc Version Clock on abort
  - (GV6) composite of GV4 and GV5
TM Design Issues

• Implementation choices
• Language design issues
• Semantic issues
Granularity

• Object
  – managed languages, Java, C#, ...
  – Easy to control interactions between transactional & non-trans threads

• Word
  – C, C++, ...
  – Hard to control interactions between transactional & non-trans threads
Direct/Deferred Update

- **Deferred**
  - modify private copies & install on commit
  - Commit requires work
  - Consistency easier

- **Direct**
  - Modify in place, roll back on abort
  - Makes commit efficient
  - Consistency harder
Conflict Detection

• Eager
  – Detect before conflict arises
  – “Contention manager” module resolves
• Lazy
  – Detect on commit/abort
• Mixed
  – Eager write/write, lazy read/write …
Conflict Detection

- Eager detection may abort transactions that could have committed.
- Lazy detection discards more computation.
Contention Management & Scheduling

- How to resolve conflicts?
- Who moves forward and who rolls back?
- Lots of empirical work but formal work in infancy
Contention Manager Strategies

• Exponential backoff
• Priority to
  – Oldest?
  – Most work?
  – Non-waiting?
• None Dominates
• But needed anyway

Judgment of Solomon
I/O & System Calls?

• Some I/O revocable
  – Provide transaction-safe libraries
  – Undoable file system/DB calls

• Some not
  – Opening cash drawer
  – Firing missile
I/O & System Calls

• One solution: make transaction irrevocable
  – If transaction tries I/O, switch to irrevocable mode.

• There can be only one ...
  – Requires serial execution

• No explicit aborts
  – In irrevocable transactions
int i = 0;
try {
    atomic {
        i++;
        node = new Node();
    }
} catch (Exception e) {
    print(i);
}
Exceptions

```
int i = 0;
try {
    atomic {
        i++;
        node = new Node();
    }
} catch (Exception e) {
    print(i);
}
```
Exceptions
Throws OutOfMemoryException!

```java
int i = 0;
try {
    atomic {
        i++;
        node = new Node();
    }
} catch (Exception e) {
    print(i);
}
```

What is printed?
Unhandled Exceptions

• Aborts transaction
  – Preserves invariants
  – Safer

• Commits transaction
  – Like locking semantics
  – What if exception object refers to values modified in transaction?
Nested Transactions

```c
atomic void foo() {
    bar();
}

atomic void bar() {
    ...
}
```
Nested Transactions

• Needed for modularity
  – Who knew that `cosine()` contained a transaction?

• Flat nesting
  – If child aborts, so does parent

• First-class nesting
  – If child aborts, partial rollback of child only
Gartner Hype Cycle

You are here

Hat tip: Jeremy Kemp
I, for one, Welcome our new Multicore Overlords …

- Multicore forces us to rethink almost everything
I, for one, Welcome our new Multicore Overlords …

• Multicore forces us to rethink almost everything

• Standard approaches too complex
I, for one, Welcome our new Multicore Overlords …

- Multicore forces us to rethink almost everything
- Standard approaches won’t scale
- Transactions might make life simpler…
I, for one, Welcome our new Multicore Overlords …

• Multicore forces us to rethink almost everything
• Standard approaches won’t scale
• Transactions might …
• Multicore programming

Plenty more to do…

Maybe you will save us…
Energy-Aware Microprocessor Synchronization: Transactive Memory vs. Locks

Tali Moeshet*, R. Iris Bahar* and Maurice Herlihy†

*Brown University, Division of Engineering, Providence, RI 02912
†Brown University, Department of Computer Science, Providence, RI 02912

tali.iris@lems.brown.edu, mph@cs.brown.edu

Abstract—One important way in which multiprocessors differ from unprocessors is in the need to provide programmers the ability to synchronize concurrent access to memory. Transactional memory was proposed as a way of improving throughput by exploiting the lower degree of synchronization conflict compared to locks. We propose a new approach to implementing transactions based on fine-grained locks, which protect relatively local locks. They do not really interfere, and the locks are difficult to use. In particular, they do not really interfere, and the locks are difficult to use. In particular, they do not really interfere, and the locks are difficult to use. In particular, they do not really interfere, and the locks are difficult to use. In particular, they do not really interfere, and the locks are difficult to use. In particular, they do not really interfere, and the locks are difficult to use.
Thanks !

תודה