NON-BLOCKING DATA STRUCTURES
AND TRANSACTIONAL MEMORY

Tim Harris, 25 Oct 2019
Lecture 3/3

- Problems with locks
- Atomic blocks and composition
- Hardware transactional memory
- Software transactional memory
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 …

No one else can make progress
Locking Relies on Conventions

- Relation between
  - Locks and objects
  - Exists only in programmer’s mind

/*
 * 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

double-ended queue

enq(x)

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-time systems
Transactions

Block of code ....

Atomic: appears to happen
  instantaneously

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

Commit: takes effect
  (atomically)

Abort: has no effect
  (typically restarted)
Atomic Blocks

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
public void LeftEnq(item x) {
    Qnode q = new Qnode(x);
    q.left = left;
    left.right = q;
    left = q;
}
public void LeftEnq(item x)
  atomic {
    Qnode q = new Qnode(x);
    q.left = left;
    left.right = q;
    left = q;
  }
}
public void LeftEnq(item x) {
    atomic {
        Qnode q = new Qnode(x);
        q.left = left;
        left.right = q;
        left = q;
    }
}
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 1\textsuperscript{st} method. If it retries...

Run 2\textsuperscript{nd} 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
Transactional Memory

committed

active

T

T

caches

memory
Transactional Memory

- committed
- active
- caches
- memory
Rewind

active

write

aborted

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 …
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
  – Try again 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
Reading an Object

Add version numbers \& values to read set
To Write an Object

Add version numbers & new values to write set
To Commit

Acquire write locks

Check version numbers unchanged

Install new values

Increment version numbers

Unlock.

Art of Multiprocessor Programming
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

Copy version clock to local read version clock

100

Shared Version Clock

100

Private Read Version (RV)
Read-Only Transactions

Read lock, version #, and memory

Copy version clock to local read version clock

Mem

Locks

100

Shared Version Clock

100

Private Read Version (RV)
Read-Only Transactions

Mem

Locks

12

32

56

19

17

Copy version clock to local

Read lock, version #, and
memory, check version #s less than

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

Shared Version Clock

Private Read Version (RV)
Read-Only Transactions

We have taken a snapshot without keeping an explicit read set!
Ordinary (read/write) Transactions

Mem

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>32</td>
<td>56</td>
<td>19</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

Locks

- Shared Version Clock
- Private Read Version (RV)

Copy version clock to local read version clock
Ordinary (read/write) Transactions

Mem

12

32

56

19

17

Locks

Copy version clock to local read version clock

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

Shared Version Clock

Private Read Version (RV)
On Commit

Art of Multiprocessor Programming 57
On Commit

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

Art of Multiprocessor Programming
On Commit

Mem

Locks

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

Mem

Locks

12
101
56
19
101
100

101
Shared Version Clock

Private Read Version (RV)
TM Design Issues

- Implementation choices
- Language design issues
- Semantic issues
Deferred/Direct 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.
Eager vs lazy conflict detection (high load)

Red-Black Tree 20% Delete 20% Update 60% Lookup
Eager vs lazy conflict detection (high load)

Red-Black Tree 5% Delete 5% Update 90% Lookup
Contention Management & Scheduling

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

Throws OutOfMemoryException!

```java
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

atomic void foo() {
    bar();
}

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

• Needed for modularity
  – Who knew that \texttt{cosine()} contained a transaction?

• Flat nesting
  – If child aborts, so does parent

• First-class nesting
  – If child aborts, partial rollback of child only
Hatin’ on TM

STM is too inefficient
Hatin’ on TM

Requires radical change in programming style
Hatin’ on TM

Erlang-style shared nothing only true path to salvation
There is nothing wrong with what we do today.
Gartner Hype Cycle

You are here

Hat tip: Jeremy Kemp
Thanks !

תודה
Overview

- Building shared memory data structures
  - Lists, queues, hashtables, ...

- Why?
  - Used directly by applications (e.g., in C/C++, Java, C#, ...)
  - Used in the language runtime system (e.g., management of work, implementations of message passing, ...)
  - Used in traditional operating systems (e.g., synchronization between top/bottom-half code)

- Why not?
  - Don’t think of “threads + shared data structures” as a default/good/complete/desirable programming model
  - It’s better to have shared memory and not need it...
Different techniques for different problems

- Ease to write
- Correctness
- When can it be used?
- How fast is it?
- How well does it scale?