Transactional Memory

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Shared Data Structures

Coarse Grained

Fine Grained

25% Shared

75% Unshared

25% Shared

75% Unshared

Fine grained parallelism has huge performance benefit
A FIFO Queue

Head

Tail

Dequeue() => a

Enqueue(d)
Coarse Grain Locks

**Simple Code**, easy to prove correct

Object lock

```
P: Dequeue() => a
Q: Enqueue(d)
```

Contention and sequential bottleneck
Fine Grain Locks

Finer Granularity, More Complex Code

Verification nightmare: worry about deadlock, livelock...
Fine Grain Locks

Complex boundary cases: empty queue, last item

Worry how to acquire multiple locks
Moreover: Locking Relies on Conventions

• Relation between
  - Lock bit and object bits
  - 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)
Lock-Free (JDK 1.5+)

Even Finer Granularity, Even More Complex Code

Worry about starvation, subtle bugs, hardness to modify...
Composing Objects

Complex: Move data atomically between structures

P: Dequeue (Q1, a)
Enqueue (Q2, a)

More than twice the worry...
Transactional Memory

“Great” Performance, “Simple” Code

Don’t worry about deadlock, livelock, subtle bugs, etc...
Promise of Transactional Memory

Don’t worry which locks need to cover which variables when...

P: Dequeue() => a
Q: Enqueue(d)

TM deals with boundary cases under the hood
Composing Objects

Will be easy to modify multiple structures atomically

Provide Composability…
The Transactional Manifesto

• Current practice inadequate
  - to meet the multicore challenge

• Research Agenda
  - Replace locking with a transactional API
  - Design languages to support this model
  - Implement the run-time to be fast enough
Transactions

• Atomic
  - Commit: takes effect
  - Abort: effects rolled back
    • Usually retried

• Serializable
  - Appear to happen in one-at-a-time order
Atomic Blocks

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

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

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

atomic {
    y = null;
}

No data race
Designing a FIFO Queue

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

Write sequential Code
Designing a FIFO Queue

```java
public void LeftEnq(item x) {
    atomic {
        Qnode q = new Qnode(x);
        q.left = this.left;
        this.left.right = q;
        this.left = q;
    }
}
```
Designing a FIFO Queue

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

Enclose in atomic block
Warning

• Not always this simple
  - Conditional waits
  - Enhanced concurrency
  - Complex patterns

• But often it is
Composition

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

Trivial or what?
Roll Back

```java
public T leftDeq() {
    atomic {
        if (this.left == null) {
            retry;
        }
        ...
    }
}
```

Roll back transaction and restart when something changes.
OrElse Composition

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

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

- Software transactional memory (STM)
- Hardware transactional memory (HTM)
- Hybrid transactional memory (HyTM, try in hardware and default to software if unsuccessful)
Hardware versus Software

• Do we need hardware at all?
  – Analogies:
    • Virtual memory: yes!
    • Garbage collection: no!
  – Probably do need HW for performance

• Do we need software?
  – Policy issues don’t make sense for hardware
Transactional Consistency

• Memory Transactions are collections of reads and writes executed atomically
• Transactions should maintain internal and external consistency
  - External: with respect to the interleavings of other transactions.
  - Internal: the transaction itself should operate on a consistent state.
External Consistency

Invariant \( x = 2y \)

Transaction A:
- Write \( x \)
- Write \( y \)

Transaction B:
- Read \( x \)
- Read \( y \)
- Compute \( z = 1/(x-y) \)
- Read \( x \)
- Read \( y \)
- Compute \( u = 1/(x-y) \)

If \( z \neq u \) then external consistency violated
Simple Lock-Based STM

• STMs come in different forms
  - Lock-based
  - Lock-free
• Here we will describe a simple lock-based STM
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 Versioned-Write-Locks

V#

V#

V#
Reading an Object

- Check not locked
- Put V#s & value in RS
To Write an Object

Mem

Locks

- Add V# and new value to WS
To Commit

- Acquire W locks
- Check V#s unchanged
  - In RS only
- Install new values
- Increment V#s
- Release ...

Mem Locks

- \( V# \)
- \( V#+1 \)
- \( V# \)
- \( V# \)
- \( V#+1 \)
Problem: Internal Inconsistency

- A Zombie is a currently active transaction that is destined to abort because it saw an inconsistent state
- If Zombies that see inconsistent states are allowed to have irreversible impact on execution state then errors can occur
- Eventual abort does not save us
Internal Consistency

Invariant $x \neq 0 \land x = 2y$

Transaction B:
- Read $x = 4$

Transaction A:
- Write $x$ (kills B)
- Write $y$

Transaction B: (zombie)
- Read $y = 4$

Compute $z = 1/(x-y)$

DIV by 0 ERROR
Solution: The “Global Clock”

- 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

- Copy V Clock to RV

Reads from a snapshot of memory. No read set!

- Check unlocked (1)
- Recheck V# unchanged (2)
  - (1)+(2) ➞ V# and mem content consistent
- Check V# < RV

100

Shared Version Clock

100

Private Read Version (RV)
Regular Transactions - during trans. prev. to commit

Mem Locks

- Copy V Clock to RV
- On read/write, check:
  - Unlocked (acquire)
  - V# < RV
  - Add to R/W set
  - (Release)

Shared Version Clock

Private Read Version (RV)
Regular Transactions - Commit

- Acquire locks (write set only)
- \( WV = \text{Fetch}\&\text{Inc}(V \text{ Clock}) \)
- For all read set
  - check unlock and
  - revalidate \( V\# < RV \)
- Update write set
- Set write set \( V\#s \) to \( WV \)
- Release locks

Mem Locks

- 12
- 100
- 56
- 19
- 100

101

69

Shared Version Clock

Private Read Version (RV)
Some explanations to lock-based implementation of regular transactions: why do we need to revalidate reads?

- When two transactions have their read and write sets intersected, but both succeed to read before the write of the other transaction occurs, then there is no way to serialize them (see example below by Nir Shavit). Hence the need to revalidate the read set *after* locking the write set.
- Also, upon commit of transaction A, *after* A already took the locks on the write set, and after or while the read set revalidated, another transaction cannot succeed to read from A’s write set before A writes it (because it is locked).

Detailed example: Take two transactions T1 and T2. Lets say that there are 2 memory locations initialized to 0. Lets say that both transactions read both locations, and T1 writes 1 to location 1 if it saw all 0’s and T2 writes 1 to location 2 if it saw all 0’s. Now if they both do not revalidate the read locations this means that T1 does not revalidate location 2 after acquiring the lock and T2 does not revalidate location 1 after grabbing the lock.

So if they both run, both read both locations, both see all 0’s in a snapshot, then both grab locks on their respective write locations, revalidate their own write locations, and write the 1 value with a timestamp greater by 1. Since they only revalidated their write locations after locking, neither saw that the other thread changed the location they only read to a 1 with a larger timestamp. Now we have a memory with two 1’s in it even though there is no such serializable execution.

Seeing a snapshot before grabbing the locks in the commit is thus not sufficient and the algorithm must have the transactions each revalidate the read set locations after acquiring the locks.
Hardware Transactional Memory

- Exploit Cache coherence
- Already almost does it
  - Invalidation
  - Consistency checking
HW Transactional Memory

read

active

T

Interconnect

caches

memory
Transactional Memory

active

read

caches

memory
Transactional Memory

committed

active

caches

memory
Transactional Memory

committed

active

caches

write

memory

memory

T

D

TD

Art of Multiprocessor Programming
45
Rewind

Caches

Memory

Aborted

Active

Write

Art of Multiprocessor Programming
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
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 transaction that could have committed.
• Lazy detection discards more computation.
Contention Management & Scheduling

• How to resolve conflicts?
• Who moves forward and who rolls back?
Contention Manager Strategies

- Exponential backoff
- Priority to
  - Oldest?
  - Most work?
  - Non-waiting?
- None Dominates

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

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

```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
Open Nested Transactions

• Normally, child commit
  - Visible only to parent
• In open nested transactions
  - Commit visible to all
  - Escape mechanism
  - Dangerous, but useful
• What escape mechanisms are needed?
Strong vs Weak Isolation

• How do transactional & non-transactional threads synchronize?
• Interaction with memory-model?
• Efficient algorithms?