Software Transactional Memory

Ellis Giles
COMP 522
Fall 2015
Outline

- Background
- Problem
- Papers
- Dynamic Software Transactional Memory (DSTM)
- Transactional Locking (TL)
- Summary
Background

- Chip Multiprocessing
- Shared Cache
- Memory Consistency Models
- Data Races
- Synchronization
- Transactional Memory
- Software Transaction Memory
Simple transfer of money between two accounts

```c
Transfer(balanceA, balanceB, amount) {
    balanceA += amount;
    balanceB -= amount;
}
```

With multiple threads, could have inconsistencies in accounts due to concurrent access

Thread 1.

```c
Transfer(AccountA, AccountB, 50);
....
```

Thread 2.

```c
....
Transfer(AccountA, AccountB, 50);
....
```

Even if Thread 2 started after Thread 1, T2 could read the original balance B and not the result after T1’s transfer. Locks are needed to guard concurrently accessed data.
Locking

- **Coarse-grained locks**
  - Lock the entire data structure
  - Effective, but poor performance

- **Fine-grained locks**
  - Lock individual data elements
  - More complex and harder to program, but has higher performance since more concurrency is possible
  - Can deadlock if order of acquisition is not consistent

```plaintext
SafeTransfer(a, b, amount)
  a.Lock; b.Lock;
  Transfer(a, b, amount);
  a.Release; b.Release;
```

```plaintext
Thread 1:
  SafeTransfer(a, b, 50);
Thread 2:
  SafeTransfer(b, a, 50);
```
Non-Blocking

Suspension of one thread cannot lead to the suspension of other threads in an algorithm

- **Wait-Free**
  - Strongest non-blocking guarantee
  - Every operation has a bound on the number of steps before completion

- **Lock-Free**
  - Allows individual threads to starve but guarantees system-wide throughput
  - When a program is run, at least one thread makes progress after sufficient time
Non-Blocking

- **Obstruction-Free**
  - An algorithm is Obstruction-Free if any thread that runs by itself for long enough makes progress without encountering a synchronization conflict
  - Ensures that a halted thread cannot prevent other threads from making progress
  - Live-lock is still possible, as conflicting threads can prevent one another from making progress
  - Allows for simpler schemes for prioritizing transactions by aborting lower priority ones
Transactions

- A sequence of steps executed by a single thread
- Generally short lived
- Similar to Database ACID guarantees for transactions
  - Atomicity, Consistency, Isolation, Durability
- Atomic – a transaction either commits (taking effect) or aborts (object is in original state)
  - All values appear to be written atomically – a challenge for software
- Should appear to execute in Isolation
  - Another thread should not be able to concurrently modify values in another thread’s transaction
  - HTM – Strict Isolation
  - STM – Weak Isolation – Memory may be referenced outside a transaction
Transactions

- **Linearizability**: single operations on single objects have a real-time order – writes should appear instantaneous on commit.

- **Serializability**: execution of sets of transactions over multiple items is equivalent to some serial execution of the transactions.

- Consistency guarantees are important:
  - Without validation, conflicts between two threads each running a transaction could cause inconsistencies.
  - Given transactions T1 & T2. T2 might modify objects in T1 that have already been accessed and modify objects that T1 will subsequently access. T1 would then only see partial effects of a transaction.
Software Transactional Memory Papers


Software Transactional Memory (STM)

- Transactional Memory was originally in hardware
- HTM has limited write set sizes
- Motivation: Software based TM would not be limited on hardware implementations, and could be unbounded in size while supporting dynamic sizes and access patterns
- STM could make concurrency programming easier using obstruction-free implementations
- Performance can be faster than traditional locking
Dynamic STM (DSTM)

- Dynamic Software Transactional Memory (DSTM) is a low level API for synchronized shared data without using locks
- Dynamic sized data structures are supported in a non-blocking, obstruction-free manner
- Objects may be created dynamically and accessed in a transaction in any order without having to list all objects accessed on start
  - E.g: `Start(X, Y, Z); x=y+z; Commit();` is unnecessary
- DSTM validates a transaction whenever a Transactional Object is opened
DSTM

- Transactional Objects - TMOBJECT
  - Container for regular objects
  - Can be created at any time
  - Values may be opened for reading or writing
    - Can throw a Denied exception
  - Must implement the TMCloneable interface.
  - Changes to objects in a transaction are visible to others on commit

- Threads - TMThread
  - beginTransaction – announced by a thread
    - Active until committed or aborted
  - commitTransaction – all writes take affect atomically
    - Returns true if the commit was successful
  - abortTransaction
DTM Example

Transfer(TMObject AccountA, TMObject AccountB, amount)
{
    thread.beginTransaction();
    try {
        balanceA = AccountA.open(WRITE);
        balanceB = AccountB.open(READ);
        balanceA += amount;
        balanceB = AccountB.open(WRITE); // UPGRADE
        balanceB -= amount;
    } catch (Denied d) {return false;}
    return thread.commitTransaction();
}

If T1 and T2 are concurrent transactions both updating AccountA and B, when T2 attempts to upgrade AccountB balance to WRITE, then could have a conflict.
Object Locator

Object Locator provides a level of indirection to access different data objects.

The **Current Version** of an object is determined by the Status of the transaction that most recently updated the object in WRITE mode.

<table>
<thead>
<tr>
<th>Transaction Status</th>
<th>Old Object</th>
<th>New Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMITTED</td>
<td>-</td>
<td>Current Version</td>
</tr>
<tr>
<td>ABORTED</td>
<td>Current Version</td>
<td>-</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>Current Version</td>
<td>Tentative Version</td>
</tr>
</tbody>
</table>

Locator

Indicates Current Version
When opening Objects across transactions, an additional level of indirection is needed to update all three Locator fields atomically.

<table>
<thead>
<tr>
<th>Transaction Status</th>
<th>Old Object</th>
<th>New Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMITTED</td>
<td>-</td>
<td>Current Version</td>
</tr>
<tr>
<td>ABORTED</td>
<td>Current Version</td>
<td>-</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>Current Version</td>
<td>Tentative Version</td>
</tr>
</tbody>
</table>
Atomically Open an Object

Compare And Swap to atomically open an object in **Transaction A** previously updated by **Transaction B**.

Indirection allows 3 fields to update atomically.
Open Object for WRITE Previously COMMITTED

For COMMITTED objects, the New Object points to Current Version.
Open Object for WRITE Previously ABORTED

For ABORTED objects, the Old Object points to Current Version.
**Open Object for WRITE Previous Transaction ACTIVE**

**Transaction A** must wait until **B** is either ABORTED or COMMITTED before proceeding. This can be resolved by a contention manager.

**Obstruction Freedom:** **Transaction A** doesn’t need to guarantee progress to **B** and can therefore attempt to abort **B**.
Open Object for READ

- Transaction A Opens an object \( o \) for reading
- Identifies the last COMMITTED version \( v \), possibly aborting the previous transaction
- Adds the pair \((o,v)\) to a thread-local read-only table
- Count is incremented for each read and decremented for each release

<table>
<thead>
<tr>
<th>O</th>
<th>V</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountA</td>
<td>Version22</td>
<td>1</td>
</tr>
<tr>
<td>AccountC</td>
<td>Version21</td>
<td>1</td>
</tr>
</tbody>
</table>
Transaction Validation

1. For each entry in the thread-local read-only table, verify that the object version is the most recently committed version
   - Another transaction could have written to the object

2. Check that the status of the Transaction is still ACTIVE
   - In case the current transaction has been aborted

• Object Open function also validates entire transaction
  • Determines which version to use
  • Transaction is validated so an inconsistent state can never be observed
Transaction Commit

- Perform Validation on the thread-local read-only table
- Attempt to use CAS to change the status field of the Transaction object from ACTIVE to COMMITTED.
Performance Costs

- Transactions opening $W$ objects for writing require $W + 1$ CAS operations - one on each Open(WRITE) call
- Cloning (copy) objects on writing
  - DSTM ensures objects are not writing during copy can use load/stores
- Validation requires $O(R)$ work
  - Must examine the thread-local read-only table of $R$ entries
- Validation must be performed to ensure consistency whenever an object is opened (for reading or writing)

**Total Work:** $O((R + W)R)$ plus $O(W)$ clone
- Transactions releasing objects opened for reading - work overhead can reduce to $O((R+W)K)$ where $K$ is the number of objects opened for reading at a time

- Synchronization conflicts can add additional work
Contestation Management

- Obstruction-Free Synchronization requires mechanisms to ensure correctness and additional mechanisms to ensure progress.
- The contenttion manager ensures progress when there are conflicts.
- Every call to a contenttion manager eventually returns and every transaction that repeatedly requests to abort another transaction eventually succeeds – obstruction freedom.
- The contenttion manager implements a ContentionManager Interface by implementing notification and feedback methods:
  - openReadAttempt is called on first read.
  - shouldAbort is called on detecting conflicting transactions.
Contestation Managers

- **Aggressive Contention Manager**
  - Always and immediately grants permission to abort any conflicting transaction
  - Still preserves correctness, as the aborted transaction will have to try again

- **Polite Contention Manager**
  - Adaptively backs off a few times when it encounters conflict
  - shouldAbort sleeps with increasing time and returns false, but after reaching a threshold returns true

- Many variations are possible
  - Application Specific
  - Advanced Contention Managers
Advanced Contention Management


- Randomized
- Karma
- Eruption
- KillBlocked
- Kindergarten
- Timestamp
- QueueOnBlock
- Polka
Integer Set

- Simple Linked List of increasing integer values
- List nodes contain the integer value and a pointer to the next element of type TMObject
- List implements the TMCloneable interface with clone
- Access a List element via a TMObject
  - `List currList = (List)prevList.next.open(READ)`
- Set is initialized with max and min values
public boolean insert(int v) {
    List newList = new List(v);
    TMObject newNode = new TMObject(newList);
    TMThread thread = (TMThread)Thread.currentThread();
    while (true) {
        thread.beginTransaction();
        boolean result = true;
        try {
            List prevList = (List) this.first.open(WRITE);
            List currList = (List) prevList.next.open(WRITE);
            while (currList.value < v) {
                prevList = currList;
                currList = (List) currList.next.open(WRITE);
            }
            if (currList.value == v) {
                result = false;
            } else {
                result = true;
                newList.next = prevList.next;
                prevList.next = newNode;
            }
        } catch (Denied d) {}
    } catch (Denied d) {
        if (thread.commitTransaction()) return result;
    }
}
public boolean delete(int v) {
    TMThread thread = (TMThread) Thread.currentThread;
    while (true) {
        thread.beginTransaction();
        boolean result = true;
        try {
            TMObject lastNode = null;
            TMObject prevNede = this.first;
            List prevList = (List) prevNede.open(READ);
            List currList = (List) prevList.next.open(READ);
            while (currList.value < v) {
                if (lastNode != null) lastNode.release();
                lastNode = prevNode;
                prevNode = prevList.next;
                prevList = currList;
                currList = (List) currList.next.open(READ);
            }
            if (currList.value != v) { result = false; }
            else {
                result = true;
                prevList = (List) prevNode.open(WRITE);
                prevList.next.open(WRITE);
                prevList.next = currList.next;
            }
        } finally {
            result = true;
        }
    }
}
Results: # Threads < # Processors

Performance of different approaches.

- Simple Locking
- IntSetSimple/Agressive
- IntSetSimple/Polite
- IntSetRelease/Agressive
- IntSetRelease/Polite
- RBTTree/Agressive
- RBTTree/Polite

Sun Fire 15K
72 Processors
# Ops in 20s

Obstruction Freedom – threads running in isolation can still make progress

Sophisticated data structures can outperform simpler ones. Livelock with an aggressive simple list. Simple locks with few threads performs well.
Results: Up to 576 Threads

Performance of different approaches.

Avoid contention whenever possible. Aggressive contention managers when threads can be pre-empted and sleep holding locks performs well.

Sun Fire 15K 72 Processors
# Ops in 20s
Results

- Livelock is possible
- Probability for contention increases with the number of processes
- It is preferable to avoid contention rather than manage it
- Releasing read objects before commit can increase performance but puts a large burden of correctness on the programmer
- More sophisticated data structures which reduce contention can outperform simpler ones
  - Red-black trees can outperform other methods due to lower levels of contention (complexity is logarithmic in set size)
- Contention manager policies play an important role in performance
- DSTM has good performance over a “Simple Lock” approach
Software Transactional Memory Papers


Transactional Locking

- **Motivation:**
  - Deadlock avoidance was suggested as the only compelling reason for non-blocking transactions (obstruction-free) – Ennals, Research at Intel ’03
  - Hand-crafted lock-based data structures tend to have the best performance
  - Design - Unbounded in addresses accessed and Dynamic in access pattern

- **Approach:** Build lock-based STM that has close to the same performance of hand-crafted lock based mechanisms with little burden on the programmer
  - Focus on small transaction performance
    - Large transactions increase vulnerability to interference
    - Large transactions have a higher probability to have I/O operations which cannot be rolled back
  - Lower the STM overhead to avoid contention
  - Use fine grained locking instead of obstruction-free data structures
Transactional Locking – Modes

- Commit Mode
  - Deferred update
  - Writes are placed in a write set
  - Locks are acquired during transaction commit
  - Highest performance – preferred mode

- Encounter Mode
  - Direct update
  - Uses an Undo-Log to rollback values
  - Acquires locks as writes are encountered
  - Releases locks after commit
  - Locks are held longer – increasing contention
TL - Versioned Lock Words

- A version locked-word is associated with every transacted memory location
- Locks are single word spinlocks: CAS acquire & store release
- Different mapping schemes are implemented and tested
  - Per Object
    - Compiler Assisted
  - Per Word
    - Also compiler assisted, but lower performance
  - Per Stripe
    - Hash variable address to lock set

Works well with HTM
TL – Data Structure and Validation

- Data Structure

- Validation
  - Walk the read-set linked list
  - Verify each Version in the linked list node matches the Version field of the associated lock – no other transaction has modified it
  - Requires time proportional to number of reads

Validating the Read-Set is a Non-Atomic Operation
TL – Commit Mode

- **Read**
  - Bloom Field checks write set and returns value if found
    - Provides linearizability avoiding Read-after-Write hazards
  - If not found, fetch lock value and record version number in read set
  - If locked can spin or abort
  - Return actual value from memory
  - Periodically validate read set

- **Write**
  - Enter variable and lock value into write set
  - Modify the same entry for writes to the same variable
1. Acquire locks of locations to be written.
   - If lock is also in read set, use CAS to acquire lock and verify version matches read set version.
   - Locks may be acquired in any order, bounded spinning is used to avoid deadlock. If bound is reached, then abort and retry.

2. **Validation:** Re-read read-locks of the read-only locations and verify version hasn’t changed
   - If values changed, release locks and abort

3. Transaction is committed

4. Release all locks in write-set
   - Atomic increment of the version number & clear lock bit (Store)
   - Note that locks were only briefly held, reducing contention.
TL – Encounter Mode

- **Stores**
  - Acquire locks to shared locations in encounter order
  - Old values and locations are saved into the write-set
  - A pointer from the lock to the write-set entry is saved
  - Write the new value to its location

- **Loads**
  - Check if the lock is free or held by current transaction (if held, get latest value)
  - If lock is held by other transaction, then bounded spin
  - Validate the read set periodically
1. **Validation**: Re-read read-locks of the read-only locations and verify version hasn’t changed
   - If values changed, release locks and abort
2. Reads are validated - transaction has committed
3. Release all locks in write-set
   - Atomic increment of the version number & clear lock bit (Store)
   - Note that locks are held longer in Encounter Mode than in Commit Mode so contention may be higher and result in weaker performance.
Handcrafted fraser-CAS skip list has 2x throughput over STM. As size of the data structure increases, contention decreases. Good scalability.
Throughput of Red-Black Tree with 5% puts and deletes and 20% puts and deletes. Mostly gets, larger structure

Performance doesn’t deteriorate.

TLCMT-PS near custom

16 Processor Sun Fire

Throughput of Red-Black Tree with 5% puts and deletes and 20% puts and deletes.
Throughput for Red-Black Tree (High Contention)

Under high contention, TL gets good scaling and performs better than others. Should use Commit Mode over Encounter Order Locking.
Summary

- Software Transactional Memory provides good performance in many cases without creating finely tuned locking of data structures.
- Using locks to implement STM shows good performance.
- Atomic Commit of multiple objects is possible.
- Validation is non-atomic.
- Overhead can increase as transactions grow in size.
- Transactions should be kept as small as possible to reduce contention.
- Different management policies on contention can affect performance.

Coming Up:
- Speculative Execution and TM on Blue Gene/Q
- Lock Elision, Transactional Memory, and Performance
- Practical Non-blocking Concurrent Objects
- Backup
Abort rates for Red-Black Trees with 20% insert and deletes and 60% gets. As size of the data structure increases, contention decreases. Abort rates don’t affect the performance of the RB Tree test very much.