Transactional Memory (TM)

CS149
Lecture 5

What’s Due

• Thursday Jan 21
  • Programming assignment 1

Outline

• Motivation: locking problems
• TM definition & key advantages
• TM programming constructs
• Caveats and open issues

Acknowledgements & Caveat

• Slides adapted from
  • TCC group at Stanford
  • Christos Kozyrakis, Ali Adl-Tabatabai, Bratin Saha (Intel)

• I’m a TM proponent
  • Might need to temper my enthusiasm

Locking Correctness Problems

• Locking is unstructured
  • Data protected by a lock is implicit
  • Locking/unlocking can be done anywhere
    • Code involved in locking not necessarily localized
    • Does not compose easily

• Races
  • Too little locking

• Deadlock
  • Bad locking discipline

Between a Lock and a Hard Place

• Locks force trade-off between
  • Degree of concurrency vs. performance
  • Chance of races, deadlock vs. correctness

• Coarse grain locking
  • Low concurrency, higher chance of correctness

• Fine grain locking
  • High concurrency, lower chance of correctness

• Is there a better synchronization abstraction?
Transactional Memory (TM)

- Memory transaction [Lomet’77, Knight’86, Herlihy & Moss’93]
  - An atomic & isolated sequence of memory accesses
  - Inspired by database transactions

- Atomicity (all or nothing)
  - At commit, all memory writes take effect at once
  - On abort, none of the writes appear to take effect

- Isolation
  - No other code can observe writes before commit

- Serializability
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed though

Programming with TM

- Declarative synchronization
  - Programmers say what but not how
  - No explicit declaration or management of locks

- System implements synchronization
  - Typically with optimistic concurrency [Kung’81]
  - Slow down only on conflicts (R-W or W-W)

```java
void deposit(account, amount){
    lock(account);
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
    unlock(account);
}
```

Advantages of TM

- Easy to use synchronization construct
  - As easy to use as coarse-grain locks
  - Programmer declares, system implements

- Performs as well as fine-grain locks
  - Automatic read-read & fine-grain concurrency
  - No tradeoff between performance & correctness

- Composability
  - Safe & scalable composition of software modules

- Failure atomicity & recovery
  - Fail fast when a thread fails
  - Failure recovery = transaction abort + restart

Example: Java 1.4 HashMap

- Fundamental data structure
  - Map: Key → Value

```
public Object get(Object key) {
    int idx = hash(key);
    HashEntry e = buckets[idx];
    while (e != null) {
        if (equals(key, e.key))
            return e.value;
        e = e.next;
    }
    return null;
}
```

Synchronized HashMap

- Java 1.4 solution: synchronized layer
  - Convert any map to thread-safe variant
  - Uses explicit, coarse-grain locking specified by programmer

```
public Object get(Object key) {  // mutex guards all accesses to map m
    synchronized(mutex) {
        return m.get(key);
    }
}
```

Concurrent HashMap (Java 5)

- Fine-grain synchronized concurrent HashMap
  - Pros: fine-grain parallelism, concurrent reads
  - Cons: complex & error prone

```
public Object get(Object key) {
    int hash = hash(key);
    Entry tab = table;
    Entry e;
    for (e = first; e != null; e = e.next) {
        if (e.hash == hash && eq(key, e.key))
            return e.value;
    }
    // Recheck under synch if key not there or interference
    Segment seg = segments[hash & SEGMENT_MASK];
    synchronized(seg) {
        tab = table;
        Entry newFirst = tab[first];
        if (e != null || newFirst != first) {
            for (e = newFirst; e != null; e = e.next) {
                if (e.hash == hash && eq(key, e.key))
                    return e.value;
            }
        }
        return null;
    }
}
```
Performance: Locks

**Balanced Tree**

![Balanced Tree Diagram]

**Hash-Table**

![Hash-Table Diagram]

Profs. Aiken/Olukotun    CS 149 Lecture 5

Performance: Locks Vs Transactions

**Balanced Tree**

![Balanced Tree Diagram]

**Hash-Table**

![Hash-Table Diagram]

TCC: a HW-based TM system

Composability: Locks

```
void transfer(A, B, amount)
synchronized(B) {
    withdraw(A, amount);
    deposit(B, amount);
}
```

• Composing lock-based code is tough
  - Goal: hide intermediate state during transfer
  - Need global locking methodology now...

• Coarse-grain locking: no concurrency
  - Between a lock and a hard place!

Composability: Transactions

```
void transfer(A, B, amount)
atomic{
    withdraw(A, amount);
    deposit(B, amount);
}
```

• Transactions compose gracefully
  - Programmer declares global intent (atomic transfer)
  - No need to know of a global implementation strategy
  - Transaction in transfer subsumes those in withdraw & deposit
    - Outermost transaction defines atomicity boundary

• System manages concurrency as well as possible
  - Serializability for transfer(A, B, $100) & transfer(B, A, $200)
  - Concurrency for transfer(A, B, $100) & transfer(C, D, $200)

Transactional HashMap

• Simply enclose all operation in atomic block
  - System ensures atomicity

```java
public Object get(Object key) {
    atomic {
        // System guarantees atomicity
        return m.get(key);
    }
}
```

• Transactional HashMap
  - Pros: thread-safe, easy to program
  - Q: good performance & scalability?
    - Depends on the implementation, but typically yes
Failure Atomicity: Locks

```java
void transfer(A, B, amount)
    synchronized(bank) {
        try {
            withdraw(A, amount);
            deposit(B, amount);
        } catch(exception1) { /* undo code 1*/
        } catch(exception2) { /* undo code 2*/
    }

    // Manually catch exceptions
    // Programmer provides undo code on a case by case basis
    // Complexity: what to undo and how...
    // Some side-effects may become visible to other threads
    // E.g., an uncaught case can deadlock the system.
```

Failure Atomicity: Transactions

```java
void transfer(A, B, amount)
    atomic {
        withdraw(A, amount);
        deposit(B, amount);
    }

    // System processes exceptions
    // All but those explicitly managed by the programmer
    // Transaction is aborted and updates are undone
    // No partial updates are visible to other threads
    // No locks held by a failing threads...
    // Open question: how to best communicate exception info
```

Programming with TM (continued)

- Basic atomic blocks: `atomic{}` ✓
- User-triggered abort: `abort`
- Conditional synchronization: `retry`
- Composing code sequences: `orelse`

User-triggered Abort

- Abort statement
  - Undo current transaction (no visible writes)
  - Jump to a specified code location
    - User Vs. system initiated abort
- Abort uses
  - Check high-level invariants in user code
  - Error and exception handling

```java
void transfer(A, B, amount)
    atomic {
        try {
            work();
        } catch(error1) { fix_code(); }
        catch(error2) { abort(); }
    }
```

Conditional Synchronization with Retry

```java
Object blockingDequeue
    // Block until queue is not empty
    atomic {
        if (isNotEmpty()) retry;
        return dequeue();
    }

    // Retry statement
    // - Rolls back current transaction
    // - Waits for change in state accessed by the transaction
    // - Everything or what specified with a watch() statement
    // - Store by another thread implicitly signals blocked thread
    // - No lost wake up compared to traditional wait-notify schemes
```

Composing Code Sequences

```java
atomic {
    q1.blockingDequeue();
} orelse {
    q2.blockingDequeue();
} orelse {
    q3.blockingDequeue();
}

    // Orelse statement
    // - Allows composition of alternative code statements
    // - If one clause fails due to retry, try next alternative
    // - Sequential order of clauses
```
TM Caveats and Open Issues

- TM Vs. Locks
- I/O and unrecoverable actions
- Interaction with non-transactional code

Atomic() ≠ Lock()+Unlock()

- The difference
  - Atomic: high-level declaration of atomicity
  - Does not specify implementation/blocking behavior
  - Does not provide a consistency model
  - Lock: low-level blocking primitive
  - Does not provide atomicity or isolation on its own

- Keep in mind
  - Locks can be used to implement atomic(), but...
  - Locks can be used for purposes beyond atomicity
  - Cannot replace all lock regions with atomic regions
  - Atomic eliminates many data races, but
  - Atomic blocks can suffer from atomicity violations
  - Atomic action in algorithm split into two atomic blocks

Example: Lock-based Code that does Not Work with Atomic

```java
//Thread 1
synchronized(lock1){
    ...
    flagB = true;
    while (flagA==0);
    ...
}

//Thread 2
synchronized(lock2){
    ...
    flagA = true;
    while (flagB==0);
    ...
}
```

- What is the problem with replacing synchronized with atomic?
- How can we code this pattern with atomic blocks?

I/O and Other Irrevocable Actions

- Challenge: difficult to undo output & redo input
  - I/O devices, I/O registers,
- Alternative solutions (open problem)
  - Buffer output & log input
  - Does not work if atomic does input after output
  - Guarantee that transaction will not abort
  - Abort interfering transactions or sequentialize the system
  - Does not work with abort(), input-after-output
  - Transaction-based systems
  - Multiple transactional devices (TM, log-based FS, ...)
  - Manager coordinates transactions across devices
    - See IBM’s QuickSilver system as a pre-TM era example

Example: Atomicity Violation

```java
//Thread 1
atomic(){
    ...
    ptr = A;
    ...
}
atomic(){
    ...
    B = ptr->field;
    ...
}

//Thread 2
atomic{
    ...
    ptr = NULL;
    ...
}
```

- What should be the transaction boundaries for the thread 1 code?

Interactions with Non-Transactional Code

- Two basic alternatives
  - Weak atomicity
    - Transactions are serializable only against other transactions
    - No guarantees about interactions with non-transactional code
  - Strong atomicity
    - Transactions are serializable against all memory accesses
      - Non-transactional loads/stores are 1-instruction transactions
- The tradeoff
  - Strong atomicity seems intuitive
  - Predictable interactions for a wide range of coding patterns
  - But, strong atomicity has high overheads for software TM
Example of Atomicity Challenges 1

- With strong atomicity
  - t1 == t2 always
  - Thread 2 may cause thread 1 transaction to abort

- With weak atomicity
  - t1 may not be equal to t2
  - Depends on exact interleaving, TM implementation, ...

//Thread 1
atomic()
{
    t1 = A;
    ...
    t2 = A;
}

//Thread 2
A++;

Example of Atomicity Challenges 2

- With strong atomicity
  - Thread 2 reads value of A before or after transaction

- With weak atomicity
  - Thread 2 may also read intermediate value
  - Depends on exact interleaving, TM implementation, ...

//Thread 1
atomic()
{
    A++;
    ...
    A++;
}

//Thread 2
t=A;

An Example without Races: Privatization

synchronized(list) {
    if (list != NULL) {
        e = list;
        list = e.next;
    }
    r1 = e.x;
    r2 = e.x;
    assert(r1 != r2);
}

Privatization on a Weakly Atomic TM

atomic{
    if (list != NULL) {
        e = list;
        list = e.next;
    }
    r1 = e.x;
    r2 = e.x;
    assert(r1 != r2);
}

atomic{
    if (list != NULL) {
        p = list;
        p.x = 9;
    }
}

Potential Solutions (Open Issue)

- Strong atomicity using hardware support
  - Full hardware TM or hardware-based conflict detection

- Optimize software overhead for strong atomicity
  - Through compiler optimizations for private and non-shared data
  - Possible for managed languages (e.g., Java); difficult for unmanaged (e.g., C++)

- Programming models that explicitly segregate transactional from non-transactional data
  - Allows correct handling of privatization & publication patterns