TRANSACTION MEMORY

Presented by
Hussain Sattuwala
Ramya Somuri

AGENDA

- Issues with Lock Free Synchronization
- Transaction Memory
- Hardware Transaction Memory
- Software Transaction Memory
- Conclusion

Issues with Lock Synchronization

- Priority Inversion
  - A high priority task is ready to execute, but a lower priority task continues execution because it holds a lock on a shared resource that the high priority task needs.
- Convoying
  - Thread holding a lock is preempted, runs out of scheduling quantum, page faults, etc. while holding a lock needed by other threads
- Deadlock
  - If one thread holding a lock dies, stalls/blocks or goes into any sort of infinite loop, other threads waiting for the lock may wait forever.

Issues with Lock Synchronization (Cont.)

- Livelock
  - Threads that need a lock are starved, unable to acquire it because other threads claim it before they get a chance
- Lock overhead:
  - The extra resources for using locks, like the memory space allocated for locks, the CPU time to initialize and destroy locks, and the time for acquiring or releasing locks. The more locks a program uses, the more overhead associated with the usage.
- Lock contention:
  - This occurs whenever one process or thread attempts to acquire a lock held by another process or thread

Issues with Lock Synchronization (Cont.)

- Composiblity
  - Locks are not composable.
- Performance problems
  - Higher performance requires more fine-grain locking
  - Can lead to more overhead and more false dependencies

Solution:

- Non-Blocking Synchronization/ Lock Free Programming

TRANSACTION MEMORY

What is a transaction?

A *Transaction is a finite sequence of instructions in a single* thread that satisfy two conditions

- Serializability
  - Transactions appear to execute serially
  - Instructions of one transaction do not interleave with another's
  - Committed transactions are never observed to execute in different orders by different processors
- Atomicity: All or nothing
  - Each transaction makes tentative changes to memory
  - When completed, a transaction commits (making changes permanent) or aborts (discarding changes) as a whole
- Isolation: Transaction executed independently

Locks versus Transactions

Lock

```
...
synchronized (lock) {
  void Toggle ()
{
  if (x == 0)
    x = 1;
  else
    x = 0;
}

}
...
```

Mapping from lock to protected data

- **lock** protects **x**

Transaction

```
...
atomic {
  void Toggle ()
{
  if (x == 0)
    x = 1;
  else
    x = 0;
}

}
...
```

Transaction protects all data

- No need to worry if another thread executes x

What is Transactional Memory (TM)?

- Transaction memory:
  - offers a mechanism that allows portions of a program to execute in isolation, without regard to other, concurrently executing tasks.
  - allows arbitrary multiple memory locations to be updated atomically and serially.

What is Transactional Memory (TM)?

Thread 1

```
begin_xaction
A = A – 20
B = B + 20
A = A – B
C = C + 20
end_xaction
```

Thread 1's accesses and updates to A, B, C are atomic

Thread 2

```
begin_xaction
C = C - 30
A = A + 30
end_xaction
```

Thread 2 sees either "all" or "none" of Thread 1's updates

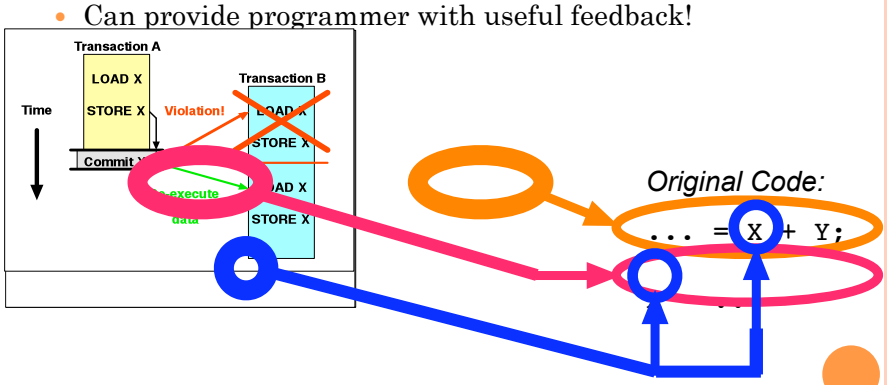
Basic mechanisms:

- Isolation: Track read and writes, detect when conflicts occur
- Version management: Record new/old values
- Atomicity: Commit new values or abort back to old values

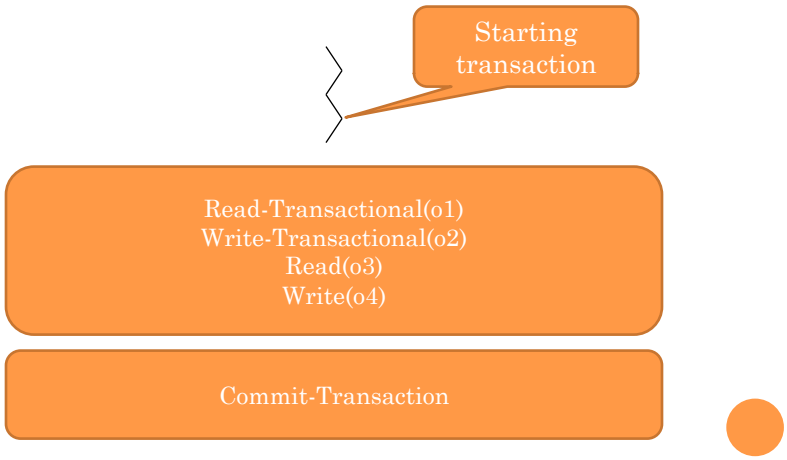
EXAMPLE 1

What if transactions modify the same data?

- First commit causes other transactions to abort & restart
- Can provide programmer with useful feedback!



THE COMPUTATION MODEL



THE COMPUTATION MODEL

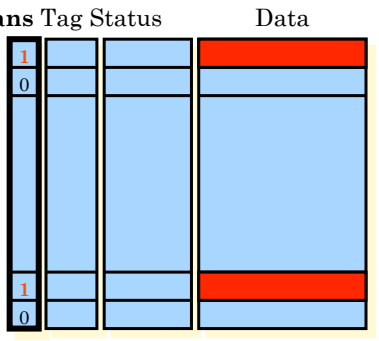
- Committing a transaction can have two outcomes:
  - Success: the transaction's operations take effect
  - Failure: the operations are discarded

IMPLEMENTATIONS

- Two implementations:
  - HTM: Hardware Transaction Memory
  - STM: Software Transaction Memory

HARDWARE TRANSACTIONAL MEMORY

- HTM: A new multi processor architecture with parallel programming mechanism that allows a group of load (read) and store (write) instructions to execute in an atomic way.
- Implemented in using the cache and cache coherency mechanism. [Herlihy & Moss]
- Uncommitted transactional data is stored in the cache.
- HTM has very low overhead but has size and length limitations.



- Trans : To determine data is transactional or/not
- Tag :

Name	Meaning
EMPTY	contains no data
NORMAL	contains committed data
XCOMMIT	discard on commit
XABORT	discard on abort

- Status : To determine if the transaction is committed or aborted

HTM PRIMITIVES(CONTD.)

For accessing memory

- Load Transactional (LT)
- Load Transactional Exclusive (LTX)
- Store Transactional (ST)

For manipulating transaction state

- Commit (COMMIT)
- Abort (ABORT)
- Validate (VALIDATE)

BRIEF DESCRIPTION OF INSTRUCTION SET

- LT : reads value of a shared memory location to a private register
- LTX : reads value of a shared memory location to a private register with the intent to write
- ST : tentatively writes a value from a private register to a shared memory location

Brief Description OF Instruction Set

- COMMIT : Attempt to make transaction's tentative changes permanent
- ABORT : Discards all updates to a transaction's write set
- VALIDATE : Test current transaction status

HTM PRIMITIVES

Transaction's Data Set is union of READ and WRITE sets

- READ SET : locations read by LT
- WRITE SET : locations accessed by LTX or ST

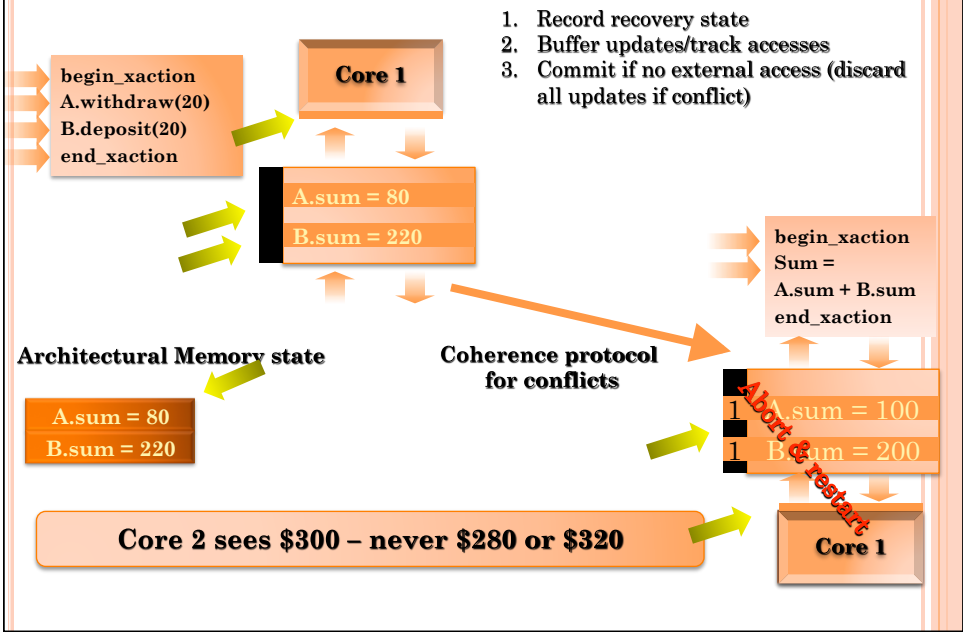
Points to be noted

- A commit succeeds if no other transaction has updated any location in the transaction's data set, and no other transaction has read any location in a transaction's write set
- If commit succeeds, all changes to write set are made visible to other threads
- If commit fails, all tentative changes to write set are discarded
- Successful validate indicates current transaction hasn't aborted (though it may later)
- Unsuccessful validate indicates a transaction has aborted, discards the transaction's tentative updates

Typical Workflow

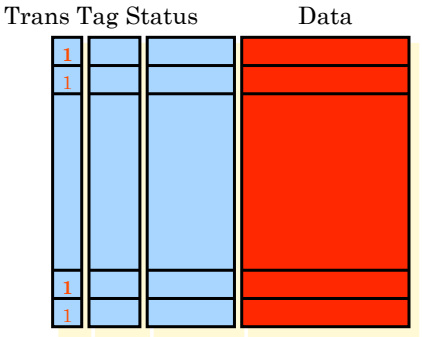
- Instead of acquiring/releasing locks around critical section, a thread can:
  - Use LT or LTX to read from a set of locations
  - Use VALIDATE to check read values are consistent
  - Use ST to modify a set of locations
  - Use COMMIT to make changes permanent
  - If either VALIDATE or COMMIT fails, ABORT and RESTART

EXAMPLE

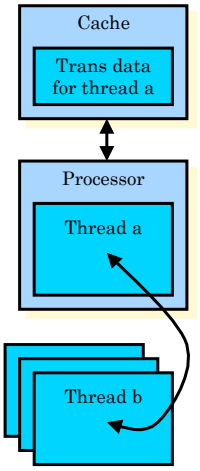


HARDWARE LIMITATION: CACHE CAPACITY

- HTM uses the cache to hold all transactional data.
- Therefore, HTM aborts transactions larger than the cache.
- Restricting transaction size is awkward and not modular.
  - Cache configuration change from processor to processor.



Hardware Limitation: Context Switches



- The cache is the only transactional buffer for all threads.
- Therefore, HTM aborts transactions on context switches.
- Restricting context switches is awkward and not modular.
  - Context switches occur regularly in modern systems (e.g.. TLB exceptions).

Software Transactional Memory

- An alternative mechanism to lock-based synchronization used to control the concurrent access to shared memory.
- Typically implemented using optimistic concurrency techniques, allowing critical Sections to proceed in parallel.

STM IN A NUTSHELL

- Start a transaction: create log
- Speculative execution—reads and writes logged
- Commit phase (atomic)
  - Read-check
  - Write to memory
- If failure, restart transaction

EXAMPLE TRANSACTION

```
native_trans<int> x = 0, y = 0;

Thread 1                          Thread 2
atomic(t) {                       atomic(t) {
  ++t.w(x);                         ++t.w(x);
  --t.w(y);                         --t.w(y);
} end_atom                        } end_atom

// CM: Thread 1's tx commits, Thread 2's tx
   aborts

// end state, x = 1, y = -1, atomic all or
   nothing
// abort if x or y change, ensure consistent
// mid-state, x = 1, y = 0 is isolated
```

Transaction terminated

Atomicity, Isolation & Consistency

```
native_trans<int> arr[100];

void read_arr(int out[])
{
  atomic(t) {
    for (int i = 0; i < 100; ++i)
      out[i] = t.r(arr[i]);
  } end_atom
}
void write_arr()
{
  atomic(t) {
    for (int i = 0; i < 100; ++i)
      t.w(arr[i]) = i;
  } end_atom
}
```

Starting State:

- arr = 0, 0, 0, ...

Ending State:

- arr = 0, 1, 2, ...

Note: read_arr()

- Never sees:
  - 0, 1, 2, 0, 0, ...
- Only: 0, 0, 0, ...
- Or: 0, 1, 2, 3, ...

Composability

- Combining atomic operations using locks—almost impossible!
- Atomic (transacted) withdrawal
  atomic { acc.Withdraw (sum); }
- Atomic deposit
  atomic { acc.Deposit (sum); }
- Atomic transfer
  atomic {
  accOne.Withdraw (sum);
  accTwo.Deposit (sum);
  }

Blocking Transactions

- Example: Producer/Consumer

```
atomic
{
   Item * item = pcQueue.Get ();
}

Item * Get () atomic // PCQueue method
{
  if (_queue.Count () == 0)
    retry;
  else
    return _queue.pop_front ();
}
```

Retry

- Restart transaction without destroying the log
- Make the read-log globally available
- Block until any of the logged read locations changes
- Every commit checks the read-sets of blocked transactions and unblocks the ones that overlap with its write-set

The Beauty of Retry

- Consumer doesn't have to specify what it's waiting for
- Producer doesn't have to signal anybody
- Composability: Wait for two items

```
atomic
{
   item1 = pcQueue.Get ();
   item2 = pcQueue.Get ();
}
```

Object-Based STM

- Transactable (atomic) objects
  - Visible as opaque handles
  - Can be opened only inside transaction
  - Open (for read) returns a const pointer to the actual object
  - Open for write clones the object and returns pointer to the clone

```
atomic struct Foo { int x; }
atomic Foo f (new Foo); // an opaque handle
atomic { // start transaction
    Foo * foo = f.open_write ();
    ++foo.x;
}
```

Cloning

- Deep copy of the object
- Embedded atomic handles are copied but not the objects they refer to
- Transactable data structures build from small atomic objects (tree from nodes)
- Value-semantic objects (e.g. structs) cloned by copy construction

Type System Support

- Struct or class marked as "atomic"—all methods (except constructor) "atomic"
- Open and open_write can be called only inside a transaction—i.e. from inside:
  - Atomic block
  - Atomic function/method
- Atomic function/method may only be called from inside a transaction

Single Linked List

```
struct Slist // not atomic
{
      this () {
            // Insert sentinels
            SNode * last = new SNode (Infin, null);
            _head = new SNode (MinusInfin, last);
      }
      // atomic methods
      const (SNode) * Head () const atomic
      {
            retrun _head.open ();
      }
      void Insert (int i) atomic;
      void Remove (int i) atomic;
private:
      atomic SNode _head; // atomic handle
}
```

List

```
struct SNode atomic
{
public:
      this (int i, const (SNode) * next) {
            _val = i; _next = next;
      }
      // atomic methods (by default)
      int Value () const { return _val; }
      const (SNode) * Next () const {
            return _next.open ();
      }
      Snode * SetNext (const (SNode) * next) {
            SNode * self = this.open_write ()
            self._next = next;
            return self;
      }
private:
      int               _val;
      atomic Snode      _next;
}
```

INSERT

```
atomic { myList.Insert (x); } // transactioned


void Insert (int i) atomic
{
      const (SNode) * prev = Head (); // sentinel
      const (SNode) * cur = prev.Next ();
      while (cur._val < i)
      {
            prev = cur;
            cur = prev.Next ();
      }
      assert (cur != 0); // at worst high sentinel
      SNode * newNode = new SNode (i, cur);
      (void) prev.SetNext (newNode);
}
```

IMPLEMENTATION

- "D" Programming Language - Bartosz Milewski
- Microsoft .NET has come up with STM in its upcoming release
- Haskell 's STM is one of the most famous.
- Intel has provided support for TM through multi-core chips

ADVANTAGES

- TM offers a simpler alternative to mutual exclusion by shifting the burden of correct synchronization from a programmer to the TM system.
- Program's author only needs to identify a sequence of operations on shared data that should appear to execute atomically to other, concurrent thread.
- Transactions make synchronization composable, which enables the construction of concurrency programming abstractions.
- Deadlock and livelock are either prevented entirely or handled by an external transaction manager; the programmer need hardly worry about it.
- Priority inversion can still be an issue, but high-priority transactions can abort conflicting lower priority transactions that have not already committed.

LIMITATIONS

- The trade-offs and programming pragmatics of the TM programming model are still not understood.
- One problem with implementing software transactional memory with optimistic reading is that it's possible for an incomplete transaction to read inconsistent state.
- The performance of TM is not yet good enough for widespread use.
  - Software TM systems (STM) impose considerable overhead costs on code running in a transaction
  - HTM fall back on software for large transactions

Conclusion

- Best approach - Lock Free Synchronization techniques.
- Relatively slow than Lock based Synchronization but No Locking Overheads.
- Lots of research still going on
- Gaining popularity in Industry

References

- Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. *ACM Conference on Principles and Practice of Parallel Programming 2005* (PPoPP'05). 2005.
- Nir Shavit and Dan Touitou (Aug 1995). Software Transactional Memory. In: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing. pp. 204--213.
- M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In International Symposium on Computer Architecture, 1993.
- "Virtualizing Transactional Memory" K. Lai et. al, ISCA 2005
- http://en.wikipedia.org/wiki/Software_transactional_memory

Thankyou!!!