Understanding the Requirements of STMs

Sathya Peri

IIT Patna

sathya@iitp.ac.in

Outline

- Motivation for Software Transactional Memory
- Drawbacks of locks
- Program Support for STMs
- Intuitive Requirements of STMs
- Formal Definition of a Correctness Criterion – Simpler Case
- Formal Definition – General Case
- Conclusion

Performance limits of single processors

- Software systems performance increase in last few decades
  - Increase in CPU clock frequencies
- Limits in the increase of clock frequencies of CPUs in the last few years
- Obtain higher performance:
  - Packing more processors into a single chip
  - They are called multi-core CPUs

Parallel Processing Applications

- To achieve better performance with multi-core CPUs
  - Applications programmed to take advantage of the underlying parallelism
- Commonly achieved by employing multi-threading
  - Multiple threads of control execute concurrently
  - Multi-threading can efficiently utilize the multi-core CPUs

Bank Transfer Example: Difficulty with parallel programming

T1	Time	T2
	1	Read B1
	2	B1 :=
B1 – 1000		
	3	Write
B1		
sum := 0	4	
Read B1	5	
Read B2	6	
sum := sum + B1	7	
sum := sum + B2	8	
	9	Read B2
"sees" wrong sum	10	B2 :=
B2 + 1000		
	11	Write
B2		

Issues with Parallel Programming

- Normally threads have to collaborate
  - Involves sharing of data in memory or on secondary storage
- Write access to shared data cannot happen in an uncontrolled fashion
  - Otherwise allows programs to see inconsistent data values
- Processors can not modify independent memory locations atomically
  - Hence, must be synchronized

Synchronization: Locks

- Traditional solution to data synchronization contention: Employing software locks
- All read and write access to shared data using locks
  - Never see inconsistent state
  - Can simulate atomic update of two different variables

Multiple Shared Data Objects

- Multiple data objects shared by programs
- Use a single program-wide lock
  - Hurts the system performance
- Multiple locks are required
  - A lock for each shared location
  - Fine grain locking

Multiple Shared Data Objects cont'd

- Threads accesses multiple shared objects
- Each thread needing access to a shared object
  - obtains the corresponding lock;
  - accesses the shared object and
  - releases the lock
- Can still result in incorrect state

Bank Transfer Example: Incorrect Locking

T1	Time	T2
	1	Lock B1
	2	Read B1
	3	B1 := B1
– 1000		
	4	Write B1
sum := 0	5	Release B1
Lock B1, B2	6	
Read B1	7	
Read B2	8	
Release B1, B2	9	
sum := sum + B1	10	
sum := sum + B2	11	Lock B2
	12	Read B2
	13	B2 := B2
+ 1000		
Again "sees" wrong sum	14	Write B2
	15	Release
B2		

Correct Locking: Two Phase Locking

- Each thread accessing shared data object
  - Locks all the shared object required first
  - Accesses the shared objects
  - Then releases all the locks
- Commonly referred to as two phase locking

Difficulties with Two Phase locking

- Two phase locking involves 'lock and wait'
- Can potentially lead to deadlocks
  - Wrong lock order
- Bank transfer example
  - T1.lock(b1) T2.lock(b2)
  - Threads are deadlocked
- Thus, obtaining locks as & when reqd will not work
  - Can still result in deadlocks

Addressing deadlocks

- Many solutions for deadlocks have been proposed in the literature
  - Deadlock prevention, deadlock avoidance, requesting resources in a non-circular manner etc.
  - Normally provided by Operating Systems
- Programmer's perspective
  - To incorporate one of the above solutions in her programs
  - Increases the complexity of programming

Software Composition

- Lock based software components are difficult to compose
- Composition: build larger software systems using simpler software components
  - Basis of modular programming
- “Perhaps the most fundamental objection [...] is that *lock-based programs do not compose*: correct fragments may fail when combined.”Tim Harris et al., "Composable Memory Transactions", Section 2: Background, pg.2

Software Composition: Difficulties with locks

- Consider a hash table with thread-safe ‘insert’ and ‘delete operations’ implemented using locks
  - Implementation of ‘insert and ‘delete’ are hidden
- User wishes to transfer an item A from hash table 1 to hash table 2
- Transfer implementation: <delete (t1, A); insert(t2, A)>
  - How to make the implementation of‘Transfer’ atomic ?

Programmer's dilemma

- The programmer is caught between two problems:
  - Increasing the part of the program that can be executed in parallel
  - Increasing the complexity of the program code and therefore the potential for problems

Alternative to locks: Software Transactional Memory

- A promising alternative which has garnered lot of interest
  - Both in Industry and Academia
- Software transactions: units of execution in memory which enable concurrent threads to execute seamlessly
  - Hide the difficulties of programming with locks
- Paradigm originates from transactions in databases

Software Transactions

- A transaction is a unit of code in execution in memory
- A software transactional memory system (STM) ensures that a transaction either
  - executes atomically even in presence of other concurrent transactions or
  - never have executed at all
- On successful completion, a transaction *commits*. Otherwise it *aborts*

Programming Support: Bank Transfer Example

```
t1()
 {
    initialization();
    atomic
    {
            Read B1
            B1 := B1 – 1000
            Write B1

            Read B2
            B2 := B2 + 1000
            Write B2
    }
}
```

Programming Support Cont'd

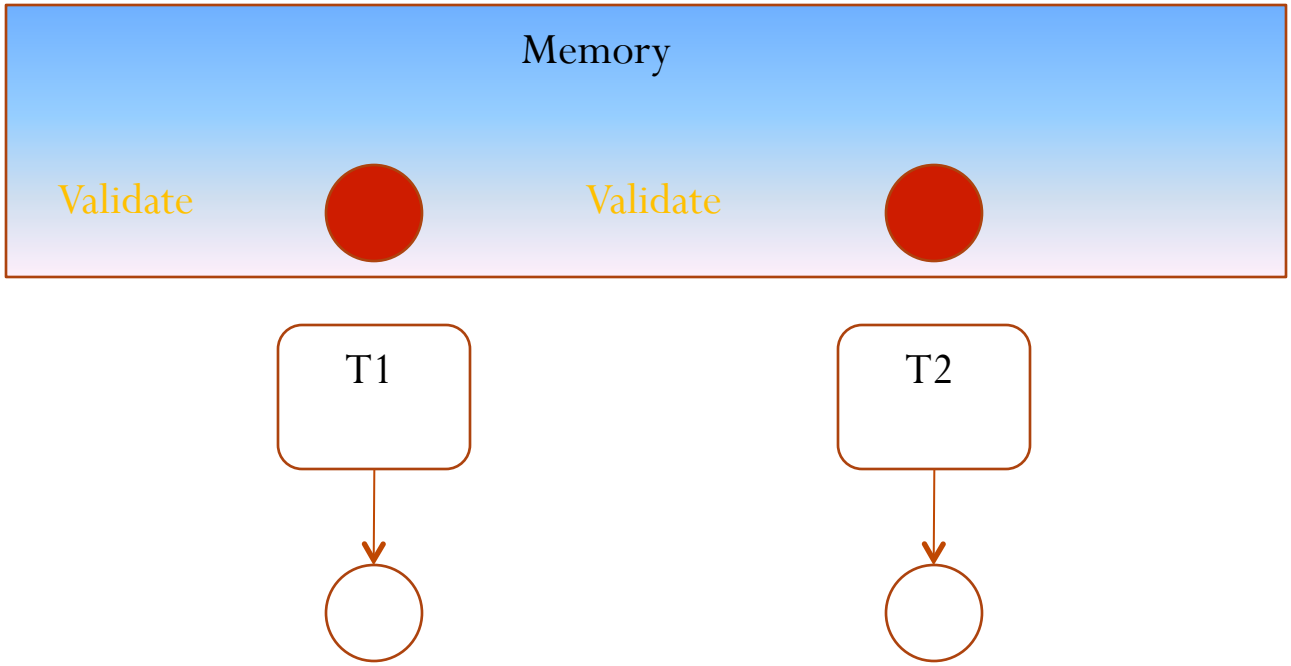
```
t2()
{
   initialization();
   atomic
   {
         Read B1;
         Read B2;

         sum := sum + B1;
         sum := sum + B2;
   }
}
```

STMs Implementation

- The operations of a transaction are divided into three phases
- Phase 1: Local read/write phase
  - Execute transaction with writes into **private buffer**
- Phase 2: Validation phase
  - Tests whether the reads and writes form a consistent view of the memory
  - Tests for conflicts
- Phase 3: Commit or Abort phase
  - Depending on validation phase, the transaction is aborted or committed
  - If committed, the local writes are performed onto the memory

Optimistic Synchronisation



Requirements of Validation Phase

- Ensures that a transaction commits only if it does not violates consistency requirements
- In the bank transfer example considering threads as transactions
  - A STM system that commits both the transactions is not semantically correct
- Hence, precise correctness criteria for STMs must be identified

Intuitive Requirements of STM Systems

- Observed by Guerraoui & Kapalka [PPoPP, 2008]
- R1: A STM system must preserve real-time order
  - Ti commits before Tj starts ➔ effects of Ti must be seen before Tj
  - violating real-time order may be misleading for programmers who are used to critical sections that enforces this order naturally
- R2: Live transactions should not affect any other transaction
  - A simple solution: Transactions execute normally. If a transaction accesses inconsistent state, abort it
  - Seems to work well in Databases

Intuitive Requirements Cont'd

- R2: Live transactions should not affect any other transaction
  - Apparently, the databases execute in controlled environment
  - Transactions accessing inconsistent states can cause problems; but they are aborted later; hence no affect
  - Consider, the following example: Initially, x=4, y=6
  - T1: z = 1/(y-x); commit; → r1(x) r1(y) c1
  - T2: x=2; y=4; commit; → w2(x) w2(y) c2;
  - Consider an interleaving
  - r1(x, 4) w2(x,2) w2(y, 4) c2 r1(y, 4)
  - z = 1/(4 – 4) → divide-by-zero error. Can cause program to crash

Intuitive Requirements Cont'd

- Reading inconsistent values can cause many kinds of errors
  - Divide-by-zero, infinite-loops, even I/O possibly
- Thus, before the system can abort a transaction that read inconsistent value
  - can cause the system to crash or
  - get into an infinite loop
- Thus, even before the inconsistent read could be detected, the damage could have already been done
- May not be acceptable for Memory Transactions

Translation into Definition – Simple Case

- What do these intuitive requirements imply?
- Consider the simple case
  - Execution consists only of read, write and tryC operations
  - All these operations are **atomic**
- Every transaction reads correct value → transactions read only from committed values
- An example history:
  - H1: r1(x, 0) w2(y, 5) w2(x, 10) r1(y, 0) c1 c2

Some Notations

- Every history has an initial committed transaction T0 that initializes all the values
- A history is said to be sequential (or serial) if all the transactions in it ordered by real-time
- **lastWrite** of a read operation r(x, v) in a history H,
  - is defined as the previous closest commit operation
  - that writes to x
- For instance, consider the earlier history H1: r1(x, 0) w2(y, 5) w2(x, 10) r1(y, 0) c1 c2
  - lastWrite of r1(x, 0): C0; lastWrite of r1(y, 0): C0

Notations continued

- Legality: a history S is said to be legal
  - if every read operation r(x,v) reads the value written by previous closest committed transaction that writes to x
- For instance:
  - H2: w1(x, 5) c1 w2(x, 10) c2 r3(x, 5) is not legal
  - H3: w1(x, 5) c1 w2(x, 10) c2 r3(x, 10) is legal

Definition of Correctness Criterion- Opacity

- The correctness criterion is called Opacity
- Consider a history H (with the restrictions as mentioned earlier)
- H is opaque if there exists a sequential (or serial) history S such that :
  - The operations of H & S are the same – (H & S are said to be equivalent)
  - S respects the real-time order of H
  - S is legal
- This definition ignores all the write steps of aborted transactions in H

Comparison of Opacity with Serializability

- Consider some examples
- H4: r1(x, 0) w1(y, 5) w2(x, 10) c1 r2(y, 5) a2
  - Serializable: T2
  - But NOT Opaque
- H5: r1(x, 0) w1(y, 5) w2(x, 10) c1 r2(y, A)
  - Opaque: T1 T2
- Serializability ignores aborted transactions. Opacity considers aborted transactions as well
- Thus, it can be seen that opacity is costlier to implement

Opacity is analogous to Strict-MVSR

- Variants of Serializability
  - View Serializability (VSR): maintains only a single-version
  - Multiversion View Serializability (MVSR): maintains multiple versions
- H6: r1(x, 0) w1(y, 5) w2(x, 10) c1 r2(y, 0) c1
  - Is in MVSR but not in VSR: T1 T2
  - MVSR allows more concurrency by storing multiple versions
- Strict MVSR: MVSR + real-time order
- Opacity: Strict MVSR + correctness of aborted transactions

Deferred write vs Direct Write Semantics

- Direct Write semantics: Transactions can read uncommitted values
- Deferred Write semantics: Transactions only read from committed values
- H7: w1(x, 5) w2(y, 10) r1(x, 5) r2(y, 10) c2 c1
  - Serializable, not Opaque: T1 T2

Verifying membership of Opacity

- Verifying membership of VSR is NP-Complete
  - Papadamitriou [JACM, 1979], Vidyasankar [AINF, 1987] showed this
- Verifying membership of Opacity?
  - Not clear if it is NP-Complete, but commonly believed to be
  - Difference is due to direct & deferred write semantics
- Membership verification is important
  - Can be used in developing efficient implementations
  - Also used in verifying correctness of implementations

Conflict Serializability (CSR)

- Conflict Order: in a history, order can be defined based on w-w, w-r, r-w operations on the same data-item
- Using this order, CSR can be defined
  - A history H is in CSR if there exists a sequential history S with the same set of conflicts
- Interestingly, membership of CSR can be verified in polynomial time using Conflict Graph (V, E)
  - V: a vertex for each transaction
  - E: based on conflict order
  - H is in CSR iff the conflict graph is acyclic
- Similarly, Conflict Opacity can be defined which can be verified efficiently

Opacity Definition– Some relaxations

- Read and Write operations are no longer atomic
  - These operations can overlap
- Operations: invocation followed by response
  - r(x) … ret(5)/ ret(A)
  - w(x, 5) …. ret(ok)
  - tryc() …. ret(ok)/ ret(A)
- Histories
  - H8: r1(x) r2(x) ret(r1(x), 0) ret(r2(x), 0) w3(y, 5) ret(w3(y, 5), ok) tryc3() r1(y) r2(y) ret(r1(y), 5) ret(r2(y), 0) ret3(ok)

Opacity Generalization

- H is opaque if there exists a sequential (or serial) history S such that :
  - The operations of H & S are the same – (H & S are said to be equivalent)
  - S respects the real-time order of H
  - S is legal
- H8: r1(x) r2(x) ret(r1(x), 0) ret(r2(x), 0) w3(y, 5) ret(w3(y, 5), ok) tryc3() r1(y) r2(y) ret(r1(y), 5) ret(r2(y), 0) ret(tryc3, ok)
  - Opaque: T2 T3 T1

Opacity Generalization

- H is opaque if there exists a sequential (or serial) history S such that :
  - The operations of H & S are the same – (H & S are said to be equivalent)
  - S respects the real-time order of H
  - S is legal
- H8: r1(x) r2(x) ret(r1(x), 0) ret(r2(x), 0) w3(y, 5) ret(w3(y, 5), ok) tryc3() r1(y) r2(y) ret(r1(y), 5) ret(r2(y), 0) ret(tryc3, ok)
  - Opaque: T2 T3 T1

Efficient Sub-Class of Opacity?

- With the generalization, it is more commonly believed that verifying opacity is NP-Complete
- Can a sub-class be defined whose membership can be efficiently verified?
- The earlier described notion of conflicts is no longer valid
  - Operations no longer atomic
  - Can new conflict notion be defined that gives efficient sub-class

Other Considerations

- Opacity is prefix-closed, whereas VSR/MVSR is not!
  - Presence of a final reading transaction
- Drawback of Opacity: interference
- Other correctness criteria: Virtual Worlds Consistency, DU-Opacity
- Nesting of Transactions: Precise correctness guarantees
- Object based STMs: Lower level conflicts do not matter. Only upper level conflicts matter.
- How is nesting different from object based STMs?

Conclusion

- Software Transactional Memory is a very promising research alternative for programming multi-core CPUs
  - Does not suffer the disadvantages of programming with locks
  - Saw the motivation
- Understood the intuitive requirements of STM systems

Questions?