Transactional Memory for Multithreaded Environments

Fall 2012
Parallel Computing
Jamar Drue
Brian Mykietka

Overview

- Jamar
  - What is Transactional Memory?
  - TM Basics
  - Hardware TM
  - Software TM
- Brian
  - Hardware Implementation
  - Software Implementation
  - Code examples
  - Conclusion
- Questions

What is Transactional Memory?

Introduction

- Shared-memory multicore microprocessors offers immense potential to exploit thread-level parallelism (TLP).

- TM was created to ease the transition from sequential algorithms to parallel algorithms for programmers.
  - Difficulties of synchronization tradeoffs, deadlock avoidance, etc.

- Simplifies concurrency programming by allowing a group of load and store instructions to execute atomically.

Previous Methods

- Parallel thread execution requires synchronization or ordering mechanisms for multiple accesses to shared data.

- Previous Multithreaded programming models
  - Use a set of low-level primitives (i.e. locks) on critical sections.
    - Guarantees mutual exclusion.
    - Ownership of one or more locks protects access to shared data.

  - Locks are complex to use and error prone.

  - With mutual exclusion locks, only one thread can hold a lock at a time.

Functionality

- Transactions replace locking with atomic execution units.
  - The programmer can focus on determining where atomicity is needed, rather than how to implement it.
  - Example atomic region in a simple kernel that computes the histogram of an array:

```
atomic {
    hist[array[i][j]]++;
}
```

  - The TM implementation determines how to run that critical section in isolation from other threads.

Functionality

- Most TM implementations assume that the transactions do not conflict, so the transactions are run in parallel.
    - If two transactions access the same memory item and at least one of them writes, then the conflict.
    - RAW dependencies are most typical.
    - If the transactions don't conflict...
        - The transactions did not have to compete lock to update the shared data.
    - If the transactions do conflict...
        - The TM must abandon (roll back) the work of one of the conflicting transactions.
        - Any attempted work must not be visible to other threads.
        - The abandoned transactions are then re-executed after the conflicts are handled.

Advantages

- TM uses mechanisms for simplifying this problem by abstracting some of these difficulties associated with concurrent access .
    - The programmer can concentrate on the algorithm instead of complex mechanisms such as locks
- With TM, multiple threads access memory simultaneously in an atomic way.
    - So either all the accesses within an atomic transaction succeed or none of the accesses succeed.
    - Shared data structures are guaranteed to be kept in consistency even in the event of a failure.
- Because actual conflicts are rare in many programs, TM takes an optimistic approach to assume that a conflict will not happen.
    - Compared to TM, locks are pessimistic.

Advantages

- Like database transaction, TM has atomicity, consistency, and isolation (ACI) properties:
  - Atomicity to guarantee transactions either commit or abort.
  - Consistency to guarantee transactions use the same total order during the whole process.
  - Isolation to guarantee that each transaction's operations are isolated to other transactions.

- TM provides a better trade-off between scaling and implementation effort.
  - Fine-grained locking scales well, but are difficult to design.

- TM is inherently deadlock free.

Disadvantages

- Disadvantages Important to Note
  - Livelock can be a problem, but it is easier to deal with than deadlock.
  - Like many high-level programming abstractions, a carefully designed algorithm using lower-level primitives can outperform an algorithm using TM.
  - Difficulty with what kind of abstractions to provide and what kind of performance tuning and debugging tools to develop for programmers.

TM Basics

Transactions

- Transaction – a sequence of instructions that either executes completely (commits) or has no effect (aborts).
  - On a successful commit, the global state is updated and all writes become visible where other transactions can use those values .
  - On an abort, the system discards all its speculative writes.
- A TM system needs a data-versioning mechanism to record the speculative writes.
  - With an Undo Log, a transaction applies updates directly to memory locations, while logging the necessary information to undo the updates in case of an abort.
  - Buffered Updates keep the speculative state in a private transaction buffer until commit time.
    - If the commit succeeds, the buffer drops the original values before the store instructions and commits the transaction's speculative stores to memory.

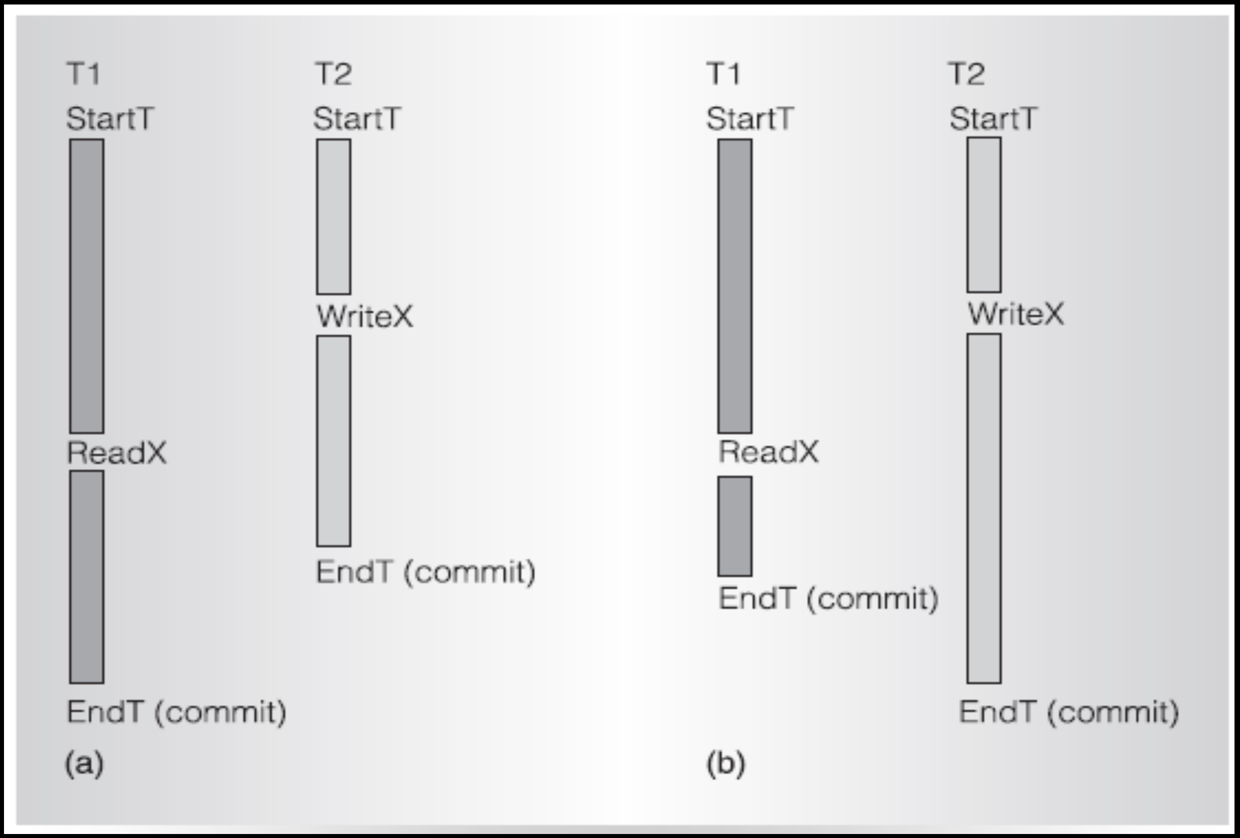
Transactions

- A transaction's instruction sequence can be explicitly or implicitly delimited.
  - Explicit
    - Some high-level programming languages include constructs that explicitly define the extent of transactions like the ''atomic'' statement shown earlier.
    - Others provide lower-level operations to explicitly start and end transactions.
    - A TM system can abort transactions explicitly by executing an abort instruction.
  - Implicit
    - In other cases, transactions start implicitly after execution of a transactional read or write operation or immediately after the commit of another transaction in the instruction stream.
    - A TM system can abort transactions implicitly because of data conflicts with concurrent transactions.

Conflict Handling

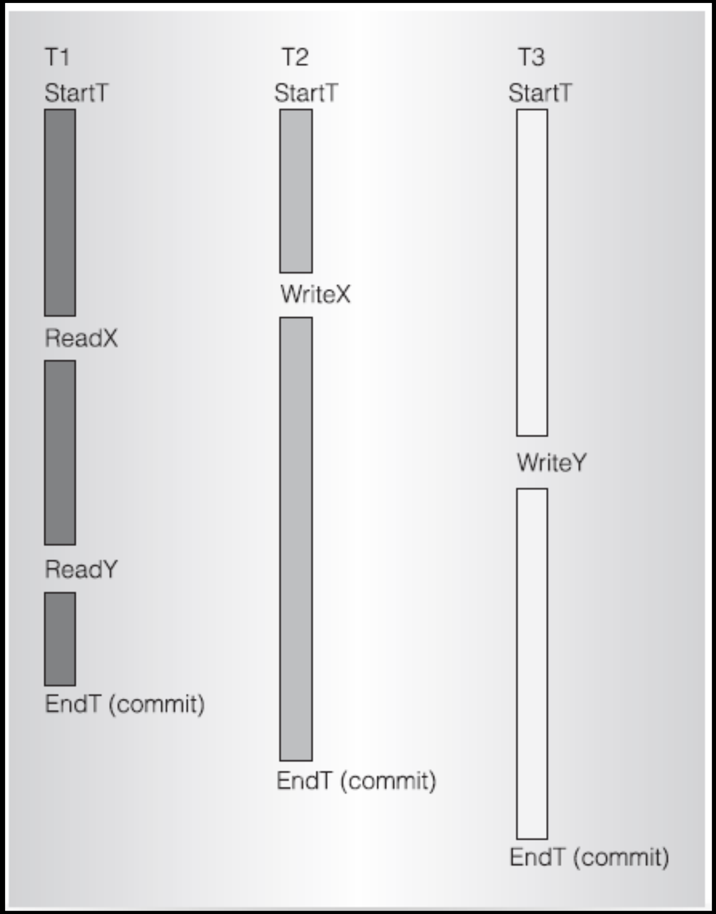
- Two issues are related to conflicts: detection and resolution.
  - Each running transaction is associated with a read set and a write set.
    - For transactional load instruction
      - memory address → read set.
    - For transactional store instruction
      - memory address + value → write set.
  - Conflict detection can be either <u>eager</u> or <u>lazy</u>.
    - Eager conflict detection checks every individual read and write for a conflict with another transaction.
    - In lazy conflict detection, a transaction checks its read and write sets for a conflict only on a commit.

Conflict Handling



Conflict Handling

- Conflict Resolution
  - Usually, a system resolves a conflict by aborting one of the transactions
  - The resolution policy has three choices
    - Committer Wins
    - Requester Wins
    - Requester stall with conservative deadlock avoidance.



TM Implementations

- Software Transactional Memory (STM)
  - Easy to implement and require no changes to existing hardware.
  - But for most STMs, poor performance and weak atomicity are two serious disadvantages.

- Hardware Transactional Memory (HTM)
  - Has the advantages of high performance and strong atomicity.
  - System architecture must support HTM.

- Combined Approach
  - Hybrid transactional memory (HyTM)
    - Supports HTM execution, but when HW resources are exceeded, falls back on STM.
  - Hardware-assisted STM (HaSTM)
    - Combines STM with new architectural support to provide STM speedup.
  - HyTM provides near-HTM performance for short transactions, while HaSTM provides performance somewhere between HTM and STM.

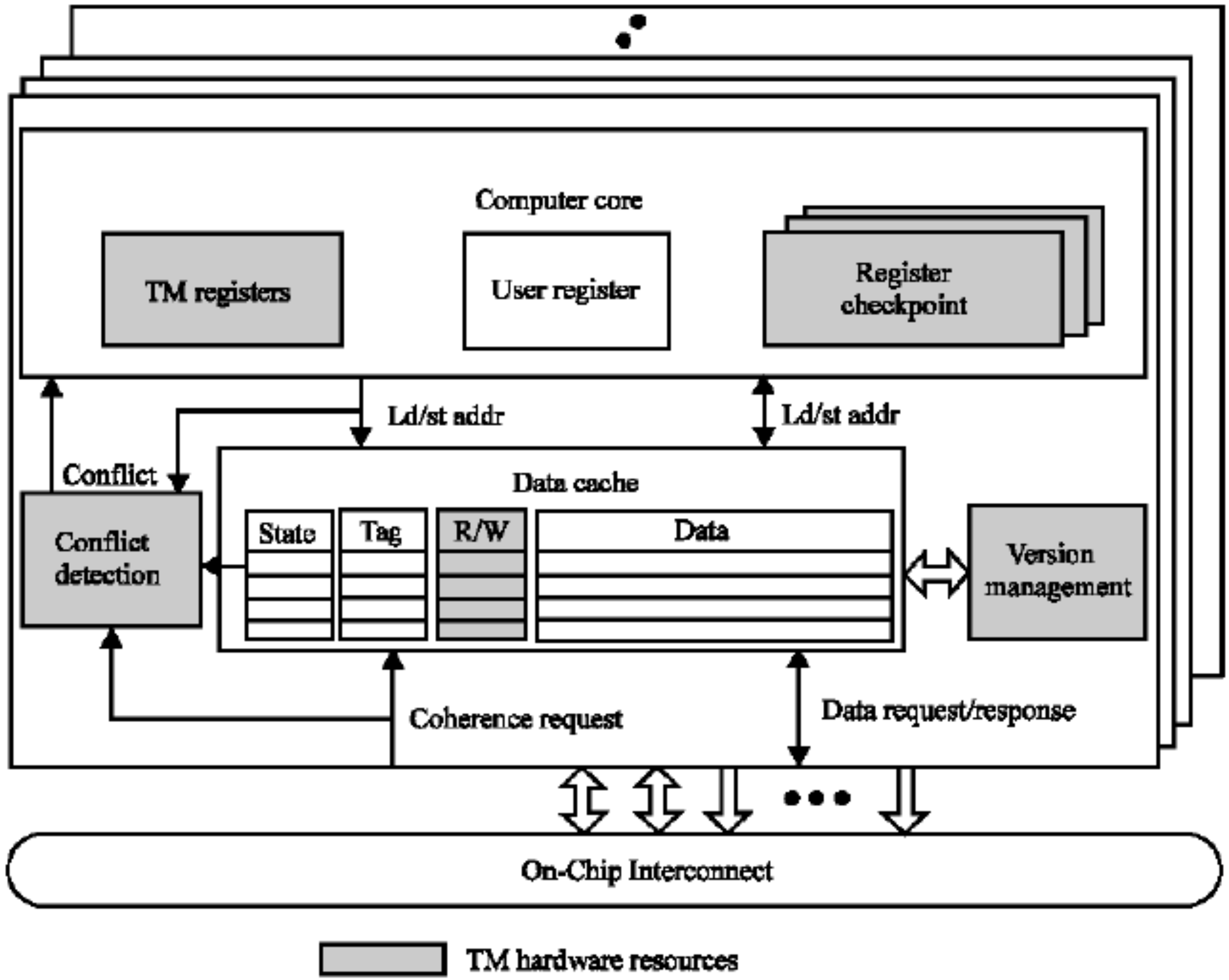
Hardware Transactional memory

Hardware TM

- The first HTM designs were minimalist
  - Modifying the cache consistency protocols
  - Complementing the ISA with new instructions.
  - Speculative state stored in extended or partitioned cache a commit or abort.

- Process
  - As a transaction starts, it checkpoints registers to save old values.
  - In order to detect read-write or write-write conflicts, memory references are tracked.
  - If a transaction completes without conflicts, its results are committed to shared memory.
  - If a conflict appears between two transactions, one of them rolls back according to register checkpoint.

- Benefits
  - HTM systems cut down the overhead of fine-grained locks.
  - They can automatically check every memory references of all the active transactions under the help of the cache coherence protocols.



Computer core
TM registers
User register
Register checkpoint
Ld/st addr
Ld/st addr
Conflict
Data cache
State
Tag
R/W
Data
Conflict detection
Version management
Coherence request
Data request/response
On-Chip Interconnect
TM hardware resources

HTM - Conflict Detection

- HTM systems keep a transaction's speculative state in the data cache or in a hardware buffer area.
  - STM systems have conflict detection at object level.
  - HTM systems work at the word or cache line level.

- The systems keep transactional loads and stores in a separate transactional cache or in conventional data caches augmented with transactional support.

- Transactional support relies on extending existing cache coherence protocols (i.e. MESI - modified, exclusive, shared, invalid), to detect conflicts and enforce atomicity.

HTM ISA Support

- ISA level transaction instructions
  - Transaction delimiters
    - start transaction (STR) .
    - end transaction (ETR).
  - Transactional Read and Writes
    - load (TLD)
    - store (TST)
  - Implicit transactions
    - When a transaction executes its first TLD or TST operation, a flag is set at the core indicating that the core is engaged in a transaction.
- Adding special instructions for abort (ABR) and validation (VLD) of a transaction makes several optimizations possible.
  - VLD allows for early conflict detection so the transaction can roll back without wasting energy.

HTM - Version Management

- The transaction’s read set and write set are stored in the data cache and keeps an extra version of the transaction’s tentative updates.
  - Two extra bits per cache line indicate whether the line is to be discarded on commit (for lines holding unmodified data) or on abort (for speculatively modified lines).

- A conflict means that a load has read invalid data and the transaction must abort.
  - The write set of the aborting transaction is dropped.

- When there is no conflict
  - The version of the original values before the store instructions are dropped.
  - The transaction’s speculative stores are committed to memory.

Software Transactional Memory

Software TM – API Design

- Software Transactional Memory (STM) has the advantages of flexibility and easy implementation.

- An STM implementation must create its own mechanism for concurrent transactions to maintain their own views of heap memory.
  - This mechanism allows a transaction to see its own writes as it runs and allows memory updates to be discarded if the transaction ultimately aborts.

- Two distinctions between how different STM systems are implemented include:
  - Transaction granularity
  - Data Organization in memory.

STM – Transaction Granularity

- Transaction granularity - the data store unit, through which a TM system detects conflicts.
  - word, block, object and hybrid.

- Word Granularity
  - A shared word is possessed by no more than one transaction at any time.
  - In order to guarantee a shared memory word to update atomically, a dedicated record is used to store the exclusive ownership of this word.

- Block Granularity
  - A multiword structure is used to store transactional variables, which include a pointer to shared data, a mutual-exclusion lock number and a wait queue used for conditional synchronization inside transactions.
  - Map shared memory addresses into a hash table, each item of which stores an ownership record for tracking whether transactions conflict.

STM – Transaction Granularity

- Object granularity,
  - With object granularity, it is unnecessary to change original object structure for translating non-transactional program to transactional program.
  - An object can execute inside and outside transactions without any change.

- Hybrid Granularity
  - In these systems, transaction granularity may change between word and object.
  - Word is used when the workload has more high-level concurrent data structures (e.g., multi-dimensional arrays)
  - Object is used when the workload has more dynamical data structures.

STM – Transaction Granularity

- Comparisons
  - Word/Block Granularity
    - Supports fine-grained sharing and fine-grained parallelism.
    - Can get more concurrently access to data structures such as array, matrix etc.
    - Provides higher conflict detection accuracy.
    - Leads to much more additional communication overhead.
    - Injures performance by making unnecessary transaction aborts.
  - Object Granularity
    - Object transactions are more helpful for supporting practical and dynamic object-based structures.
    - Hard to support object transactions for non-object.
    - High parallel data structures such as arrays, using objects for conflict detection can cause unnecessary conflicts, inhibiting concurrency.

STM - Data Management

- A high-level distinction between STM implementations is how they organize data in memory.
  - One approach separates transactional data and ordinary data, introducing a distinct memory format for transactional objects. (Indirect)
  - An alternative approach allows data to retain its ordinary structure in memory, and the STM uses separate structures to maintain its own metadata. (Direct)
- There are advantages and disadvantages to each approach.

Indirect Data Management

- Since, transactional and ordinary data are stored in different memory structure, these systems cannot access transactional data directly.
  - If a transaction wants to access a shared object, it must take actions to open a TM object first.
  - The open operations are different according to whether the access mode is READ or WRITE.
    - READ mode - the same object body can be shared by multiple transactions at the same time.
    - WRITE mode - a new version copy of the object is prepared for update and is only visible to the transaction until the transaction commits.
  - Makes transactional data semantics clear

Direct Data Management

- Transactional and ordinary data are stored in the same low-level memory structure in the system

- They refer transactional data by ordinary pointer directly.

- They are convenient for spatial access locality and hence improve performance and transaction throughput.

STM - Version Management

- STM API implementation has two ways of managing tentative updates: Buffered updates or Undo log.
  - Buffered updates/Lazy Version Management (LVM)
    - A transaction keeps a private shadow copy of all the memory words it updates.
    - STMRead accesses the shadow copies so that they will see earlier writes by the same transaction.
    - Hashing maps an address to a slot in the current transaction’s shadow table.
    - Benefits
      - LVM is more efficient for transactions aborting.
      - LVM allows concurrent transactional read and write for the same logical data.
      - Keeping a private version of the object in store buffer and no one committing at the time.

STM - Version Management

- Undo-log/Eager Version Management (EVM).
  - STMWrite directly updates the heap so that calls to STMRead will see earlier updates without needing to search a table.

  - STMWrite maintains an undo log of all values that it overwrites referred to as checkpoints
    - On commit, discard the old version in its checkpoint.
    - On an abort, the old version in its checkpoint is restored to its original place and the new version is discarded.

  - Benefits
    - VM is more efficient for transactions committing.

  - Disadvantages
    - Prevents other transactions to read a modified uncommitted object, limiting possible concurrency.

Conflict Detection

- Generally, there are three type of conflict detection: Eager Conflict Detection (ECD), Lazy Conflict Detection (LCD) and Hybrid Conflict Detection (HCD).

- ECD
  - Detects conflicts when a transaction wants to access memory.
  - ECD always works with EVM, since it is necessary to make sure that only one transaction can write a new version to a logical data.
- LCD
  - Detects conflicts when a transaction is about to commit updates
  - Similarly, LCD commonly works with LVM.

- HCD, combines ECD and LCD.
  - Manage transactional version with EVM mechanism.
  - Uses ECD before a transaction read or write.
  - Allow multiple transactions to read a shared data concurrently and to delay detecting conflicts until committing with LCD.

Synchronization

- Synchronization is the mechanism to guarantee that a transaction attempting to access a logical data will finish its work.
  - Blocking Synchronization (BS)
  - Non-blocking Synchronization (NS).

- The BS blocks concurrent access
  - In order to keep consistency, BS forces multiple threads to access critical sections exclusively, maintaining a queue in the order of request(wait-state).

  - A compiler can automate this approach, by using locks as a transaction executes until it commits.

  - Disadvantages
    - This wait-state easily leads to severe problems such as deadlock, priority inversion, contention, etc.

Synchronization

- NS prevents concurrent threads from entering wait-state.
  - In NS, a concurrent thread may either abort its transaction, or abort the transaction of conflicting thread.

  - The NS has been classified into three main categories based on their assurances for forward progress:
    - Wait-freedom
      - Assures all threads avoid deadlocks and starvation.
    - Lock-freedom
      - Assures all threads avoid deadlocks, but not starvation.
    - Obstruction-freedom
      - Assures all threads avoid deadlocks, but not livelocks .
      - Livelock can be effectively minimized with simple methods like exponential backoff.

  - Disadvantages
    - NS may cause more memory traffic than BS.

Existing Implementations

Existing Implementations

- Hardware implementations
  - Sun - Rock microprocessor
  - IBM Blue Gene/Q
  - IBM zEnterprise EC12
  - Transactional Synchronization Extensions (TSX)
- Software implementations
  - Code examples
    - C/C++ Boost.STM
    - C# SXM

Hardware Implementations

- Sun - Rock microprocessor (2006 - 2009)
  - First production processor to support transactional memory
  - Added two new instructions `chkpt` and `commit` and one new status register `cps`
  - `chkpt <fail_pc>` used to begin transaction
  - `commit` to commit transaction
  - If transaction aborts then we jump to `<fail_pc>` and `cps` is used to determine reason

Hardware Implementations

- Sun - Rock microprocessor
  - Transactional memory support is best-effort based
    - Does not guarantee support of transactions of any size
    - Committed in in-cache and aborted if don't fit
  - Transactions can be aborted for other reasons
    - TLB misses
    - Interrupts
    - Certain commonly used function call sequences
    - "Difficult" instructions (division)

Hardware Implementations

- Blue Gene/Q processor (2012) (Ranked #2 - top500.org)
  - L2 multi-versioned, transactional memory and speculative execution, hardware support for atomic operations
  - Implemented in hardware, can access all memory up to 16GB boundary
  - Transactions implemented through regions of code that are designated as single operations
  - These regions are called transactional atomic regions

Hardware Implementations

- Blue Gene/Q processor - Transactional memory
  - When transactional memory is activated, transactions run in one of two modes
    - Speculation mode
      - Allows for coarse grain multi-threading
      - load/store conflicts detected and resolved according to sequential semantics
      - Long running speculation mode (default)
      - Short running speculation mode
    - Irrevocable mode
  - Each mode applies to an entire transactional atomic region

Hardware Implementations

- Blue Gene/Q processor - Execution modes
  - Speculation mode
    - Kernel address space, devices I/O, memory-mapped I/O are protected from irrevocable actions
    - Transaction goes into irrevocable mode if such an action occurs to guarantee correct results
  - Irrevocable mode
    - System calls, irrevocable operations such as I/O operations, and OpenMP constructs trigger transactions to go into speculation mode which serializes the transactions
    - Transactions run in this mode when max number of transaction rollbacks has been reached
    - Each memory update of thread is committed instantaneously instead of at end of transaction → memory updates immediately visible to other threads

Hardware Implementations

- Blue Gene/Q processor - Built-in transactional memory functions
  - Can create struct to fill out fields:
    - Hardware thread ID
    - Total number of transactions
    - Total number of rollbacks for transactional memory threads
    - Various other serialization counts
  - This struct can be passed into functions to be populated:
    - `tm_get_stats(TmReport_t *stats)`
    - `tm_get_all_stats(TmReport_t *stats)`
  - Can also call write statistics for transactional memory of particular hardware thread to a log file using:
    - `tm_print_stats()`
    - `tm_print_all_stats()`
  - `#pragma tm_atomic` specifies atomic region

Hardware Implementations

- Transactional Synchronization Extensions (TSX)
  - Extension to the x86 ISA that adds HTM support
  - Documented by Intel in February 2012 scheduled for implementation in microprocessors based on Haswell architecture
  - Hardware monitors multiple threads for conflicting memory accesses and aborts/rolls back transactions that cannot complete successfully

Hardware Implementations

- Transactional Synchronization Extensions (TSX)
  - Programmer has ability to specify code regions to be executed transactionally
  - Provides two software interfaces to specify regions:
    - Hardware Lock Elision (HLE)
      - Legacy `XACQUIRE`/`XRELEASE` instructions
      - Allows optimistic execution by suppressing the write to lock so lock appears to be free to other threads
      - Failed transaction restarts from `XACQUIRE`
    - Restricted Transactional Memory (RTM)
      - New instruction set interface
      - `XBEGIN`, `XEND`, `XABORT` instructions
      - Allows programmers to define transactional regions in more flexible manner than with HLE
      - Gives programmer ability to specify fallback code path

Software Implementations

Software Implementations

Proposed Language Support

- Simplest form "atomic block"

```
// Insert a node into a doubly linked list atomically
atomic
{
    newNode->prev = node;
    newNode->next = node->next;
    node->next->prev = newNode;
    node->next = newNode;
}
```

- When end of block reached,
  - Transaction committed if possible
  - Or else aborted and retried

Software Implementations

Proposed Language Support

- Conditional critical region (CCR) permit guard condition

```
atomic (queueSize > 0)
{
    // remove item from queue and use it
}
```

- Enables transaction to wait until it has to do work
- If condition is not satisfied, transaction manager will wait until another transaction has made a commit that affects the condition before retrying

Software Implementations

Proposed Language Support

- Composable Memory Transactions, adds retry command
- Can abort transaction at any time and wait until some value previously read by the transaction is modified before retrying

```
atomic
{
     if (queueSize > 0)
     {
          // remove item from queue and use it
     }
     else
     {
          retry
     }
}
```

Software Implementations

- Currently a hot area of research
- Many implementations are still considered experimental
- Numerous implementations in various languages:
  - C/C++
  - C#
  - Clojure
  - Common Lisp
  - Haskell
  - Java
  - JavaScript
  - OCaml
  - Perl
  - Python
  - Scala
  - Smalltalk

Source: http://en.wikipedia.org/wiki/Software_transactional_memory

Software Implementations

Various C/C++ Implementations

- TinySTM - time-based STM, integrates STM with C/C++ with LLVM
- LibCMT - open-source implementation based on "Composable Memory Transactions"
- Intel STM Compiler Prototype Edition
  - Implements STM for C/C++ directly in compiler producing 32 or 64 bit code for Intel or AMD processors
  - Implements `atomic` keyword
  - Provides ways of decorating (declspec) function definitions to control/authorize use in atomic sections
  - This is a substantial implementation with the stated purpose to enable large scale experimentation in any C/C++ program

Software Implementations

C/C++ Implementation

- Boost.STM - Library under construction
  - Optimistic concurrency
  - ACI transactions
    - Atomic - all operations execute or none do
    - Consistent - only legal memory states
    - Isolated - other transactions cannot see until committed
  - Language-like `atomic` transaction macro blocks - like above
  - Closed, flattened composable transactions
  - Direct and deferred updating run-time policies
  - Validation/invalidation conflict detection policies
  - Lock-aware transactions
  - Programmable contention management
  - Isolated/irrevocable transactions for transactions that must commit

Software Implementations

C/C++ Implementation

- Boost.STM "Hello World" example
  - Both read and write on `counter` variable function atomically or neither operations are performed
  - Transaction begins and ends in legal memory states
  - Intermediate state of incremented `counter` is isolated until the transaction is complete

```
#include <boost/stm.hpp>
Boost::stm::tx::object<int> counter(0);

int increment() {
    BOOST_STM_TRANSACTION {
        return counter++;
    } BOOST_STM_TRANSACTION;
}
```

Software Implementations

C/C++ Implementation

- Boost.STM - Simple Transaction Example - Linked List Insert
  - tx_ptr smart pointer
  - 100 atomic insertions
  - No additional code needed to perform transactional linked list
  - Simple!

```
tx_ptr< linked_list<int> > linkedList;
...
for (int i = 0; i < 100; ++i) {
    BOOST_STM_TRANSACTION {
        linkedList->insert(i);
    } BOOST_STM_TRANSACTION;
}
```

Software Implementations

C/C++ Implementation

- Boost.STM - Insert Retry Transaction Example
  - Code performs two key operations
    1. Retries the transaction until it succeeds (commits)
    2. Catches aborted transaction exceptions
  - aborted_transaction_exception - exception neutral while gaining performance benefits from early notification of doomed transactions

```
void insert(T const &val)
{
    BOOST_STM_TRANSACTION
    {
        // our code to insert
    } BOOST_STM_END_TRANSACTION;
}
```

Software Implementations

Various C# Implementations

- SXM - Implemented by Microsoft Research
- NSTM - .NET STM, truly nested transactions and integrating with System.Transactions
- MikroKosmos
  - Verification-oriented model implementation of STM (Bartok STM)
  - Implementation meant for benchmarking, not practical use
- STM.NET
  - Microsoft DevLabs project
  - Delineate sections of code as running with an atomic block using a delegate or try/catch

Software Implementations

C# Implementation

- SXM Overview
  - Facilitate experimentation with new algorithms and techniques for implementing STM
  - Users encouraged to implement/experiment with new components
    - Benchmarks
    - Contention managers
      - Greedy - Maximal independent set running
      - Aggressive - Always aborts conflicting transactions
      - Priority - Prior transaction has later timestamp, abort it
    - Object factories

Software Implementations

C# Implementation - SXM

```
[Atomic]
public class Node
{
    protected int value;
    protected Node next;
    public Node(int value)
    {
        this.value = value;
    }
    public virtual int Value
    {
        get { return value; }
        set { this.value = value; }
    }
    public virtual Node Next
    {
        get { return next; }
        set { this.next = value; }
    }
}
```

Software Implementations

C# Implementation - SXM

- Factory creates transactional proxies that intercept property calls:

```
IFactory factory = new XAction.MakeFactory(typeof(Node));
```

- Can create Node objects by using:

```
Node node = (Node)factory.Create(value);
```

Software Implementations

C# Implementation - SXM

```
public override object Insert(object _v)
{
    int v = (int)_v;
    Node newNode = (Node)factory.Create(v);
    Node prevNode = this.root;
    Node currNode = prevNode.Next;
    while (currNode.Value < v)
    {
        prevNode = currNode;
        currNode = prevNode.Next;
    }
    if (currNode.Value == v)
    {
        return false;
    }
    else
    {
        newNode.Next = prevNode.Next;
        prevNode.Next = newNode;
        return true;
    }
}
```

Software Implementations

C# Implementation - SXM

- To prepare method to be executed by transaction, turn it into an XStart delegate

```
XStart insertXStart = new XStart(Insert);
```

- To execute the transaction:

```
XAction.Run(insertXStart, value);
```

Software Implementations

C# Implementation - SXM

- Conditional Waiting
  - XAction.Retry()
  - Aborts current transaction, restarts it when some object accessed by that transaction has been modified
- OrElse Combinator
  - Provides way to specify alternative execution paths
  - Example
    - Remove item from buffer b1, but buffer is empty
    - Instead of blocking you would prefer to remove an item from buffer b2
    - Get1() - remove item from b1, Get2() - remove from b2

```
getXStart = XAction.OrElse(new XStart(Get1), new XStart(Get2));
int x = (int)XAction.Run(getXStart);
```

Conclusion

- Great alternative to lock-based synchronization
- Simplifies conceptual understanding of multi-threaded programs, makes programs more maintainable by working in harmony with high-level abstractions such as objects and modules
- Many implementations, each with own strengths and weaknesses
- Beginning to see more mainstream interest in TM with multi-threaded applications being much more prevalent

References

1. Harris, T.; Cristal, A.; Unsal, O.S.; Ayguade, E.; Gagliardi, F.; Smith, B.; Valero, M.; , "Transactional Memory: An Overview," *Micro, IEEE* , vol.27, no.3, pp.8-29, May-June 2007
2. Wang, X., Z. Ji, C. Fu and M. Hu, 2010. A review of software transactional memory in multicore processors. Inform. Technol. J., 9: 192-200
3. Wang, X., Z. Ji, C. Fu and M. Hu, 2009. A review of hardware transactional memory in multicore processors. Inform. Technol. J., 8: 965-970.

References

- http://en.wikipedia.org/wiki/Transactional_memory
- http://en.wikipedia.org/wiki/Software_transactional_memory
- http://en.wikipedia.org/wiki/IBM_zEC12_(microprocessor)
- http://www.eetimes.com/electronics-news/4218914/IBM-plants-transactional-memory-in-CPU
- http://pic.dhe.ibm.com/infocenter/compbg/v121v141/index.jsp?topic=%2Fcom.ibm.xlcpp121.bg.doc%2Fproguide%2Fbg_tm_concept.html
- http://en.wikipedia.org/wiki/Rock_processor
- http://www.dolcera.com/wiki/index.php?title=Transactional_memory
- https://svn.boost.org/trac/boost/wiki/LibrariesUnderConstruction
- http://svn.boost.org/svn/boost/sandbox/stm/branches/vbe/libs/stm/doc/html/toward_boost_stm/users_guide/getting_started.html
- http://svn.boost.org/svn/boost/sandbox/stm/branches/vbe/libs/stm/doc/html/toward_boost_stm/users_guide/tutorial.html
- http://en.wikipedia.org/wiki/Transactional_Synchronization_Extensions
- http://software.intel.com/sites/default/files/m/3/2/1/0/b/41417-319433-012.pdf
- http://research.microsoft.com/en-us/downloads/c282dbde-01b1-4daa-8856-98876e513462/
- http://www.cs.brown.edu/~mph/SXM/README.doc
- ftp://ftp.research.microsoft.com/downloads/fbe1cf9a-c6ac-4bbb-b5e9-d1fda49ecad9/SXM1.1.zip
- http://blogs.msdn.com/b/stmteam/archive/2009/07/28/stm-net-released.aspx
- http://www.disco.ethz.ch/lectures/fs11/seminar/paper/johannes-2-2.pdf
- http://en.wikipedia.org/wiki/Simultaneous_multithreading
- ftp://public.dhe.ibm.com/common/ssi/ecm/en/dcw03006usen/DCW03006USEN.PDF