Portland State University
ECE 588/688

Transactional Memory

Issues with Lock Synchronization

- Priority Inversion
  - A lower-priority thread is preempted while holding a lock needed by higher-priority threads
- Convoying
  - Thread holding a lock is preempted, runs out of scheduling quantum, page faults, etc. while holding a lock needed by other threads
- Deadlock
  - Processes attempt to lock the same set of objects in different orders, cyclic dependence

Deadlock Example

- Concurrent Bank transfers

Transfer 1 (from A to B)	Transfer 2 (from B to A)
Lock(A)	Lock(B)
Load(A)	Load(B)
Lock(B)→locks A, fails to lock B	Lock(A) →locks B, fails to lock A
Load(B)	Load(A)
Calculate new value for A	Calculate new value for B
Calculate new value for B	Calculate new value for A
Write(A)	Write(B)
Unlock(A)	Unlock(B)
Write(B)	Write(A)
Unlock(B)	Unlock(A)

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
- False inter-thread dependencies
  - Conservative programming style can lead to thread serialization, even if it is not really needed
- Performance problems
  - Higher performance requires more fine-grain locking
  - Can lead to more overhead and more false dependencies

Back to Deadlock Example

Transfer 1 (from A to B)	Transfer 2 (from B to A)
Begin Transaction	Begin Transaction
Load(A)	Load(B)
Load(B)	Load(A)
Calculate new value for A	Calculate new value for B
Calculate new value for B	Calculate new value for A
Write(A)	Write(B)
Write(B)	Write(A)
End Transaction	End Transaction

- When there are no conflicts, both transactions complete successfully
- When there is a conflict (above), one transaction commits and the other aborts

Solution: Lock-Free Synchronization using Transactional Memory

- Transactional Memory
  - Allows programmers to define customized Read-Modify-Write operations that apply to multiple words of memory
  - Implemented by extending cache coherence protocols
- A *Transaction* is a finite sequence of instructions in a single thread that satisfies 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

Related Concept: Database Transactions

- Transactions are a widely used concept in database systems
- A database transaction satisfies the ACID properties:
  - Atomicity: Transaction is executed as a whole, or no part of it is executed (similar to last slide)
  - Consistency: If database is in a consistent state before transaction, it should be consistent after transaction
  - Isolation: Concurrent transactions will not interfere with each other's execution. Intermediate changes by a transaction are not seen outside transaction until transaction is committed
  - Durability: After commit, a transaction's changes are permanent even when system fails
- When a conflict occurs, some transactions are killed to allow others to commit

Transactional Memory Concepts

- TM primitives
  - Load-Transactional (LT): reads value of a shared memory location to a private register
  - Load-Transactional-Exclusive (LTX): reads value of a shared memory location to a private register with the intent to write
  - Store-Transactional (ST): Tentatively writes a value from a private register to a shared memory location
- Read and write sets
  - Read set: locations read by LT
  - Write set: locations accessed by LTX or ST
  - Transaction's data set: Union of read and write sets

Changing A Transaction's State

- COMMIT: Attempt to make transaction's tentative changes permanent
  - 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
- ABORT: Discards all updates to a transaction's write set
- VALIDATE: test current transaction status
  - Successful validate indicates current transaction hasn't aborted (though it may abort later)
  - Unsuccessful validate indicates a transaction has aborted, discards the transaction's tentative updates

Suggested Use for Transactions

- 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
- Can be implemented in software, but hardware implementation is needed for good performance
- Hardware support implies limited transaction size
  - May trap to software on overflow

Hardware Implementation Guidelines

- Non-transactional operations use the same caches, cache controllers, and coherence protocols that they would've used in the absence of TM
- Custom hardware support restricted to L1 caches and instructions that communicate with them
- Committing or aborting a transaction is a local operation to the cache, doesn't require communicating with other threads or writing data back to memory

Example Implementation

- Extends Write-Once snooping coherence protocol
- Each processor maintains two caches
  - Regular cache for non-transactional operations (direct-mapped)
  - Transactional cache for transactional operations (fully associative)
    - Similar to Jouppi's Victim cache
    - Holds all tentative writes without propagating them to other processors or memory unless the transaction commits
- Cache Line States: Paper Tables 1 and 2
  - XCOMMIT lines contain old data, XABORT lines contain tentatively modified data
  - On Commit, XCOMMIT entries discarded, XABORT entries change to NORMAL
  - On Abort, XABORT entries discarded, XCOMMIT entires change to normal
- Bus transactions: Paper Table 3

Example Implementation: Processor Actions

- Processor maintains two flags
  - Transaction active (TACTIVE): Whether a transaction is in progress
  - Transaction status (TSTATUS): Whether transaction is active or aborted
- Non-transactional operations behave like original coherence protocol
- Transactional operations issued by aborted transaction cause no bus cycles, may return arbitrary values
- VALIDATE inst. returns TSTATUS flag
  - If false, sets TACTIVE to false and TSTATUS to true
- ABORT inst. sets TSTATUS to true and TACTIVE to false
- COMMIT returns TSTATUS, sets TSTATUS to true and TACTIVE to false

Example Implementation: Processor Actions (Cont.)

- LT issued by active transaction
  - Probe Transactional cache for an XABORT entry and return its value.
  - If hit to NORMAL entry, it changes to XABORT, and an XCOMMIT entry is allocated
  - If no NORMAL or XABORT entries exist in transactional cache, issue T_READ cycle on bus. When it completes successfully, set up one XABORT and one XCOMMIT entry in transactional cache
  - If T_READ returns BUSY, abort transaction (TSTATUS ← false, drop all XABORT entries, set XCOMMIT entries to NORMAL)
- LTX issued by active transaction
  - Uses T_RFO on miss (instead of T_READ)
  - Change cache state to RESERVED if T_RFO succeeds
- ST issued by active transaction
  - Similar to LTX except that it updates the XABORT entry's data

Example Implementation: Cache Actions

- Both regular and transactional caches snoop bus
  - Ignore all requests for addresses not in the cache
- Regular cache actions
  - READ or T_READ: If state is VALID, return value. If state is RESERVED or DIRTY, return value and reset state to VALID
  - RFO or T_RFO: return data and invalidate own line
- Transactional cache actions
  - Acts like regular cache if TSTATUS is false or a request is non-transactional (READ or RFO), except that it ignores entries with transactional tags other than NORMAL
  - T_READ: If state is VALID, return value
  - All other transactional operations: Return BUSY
- Memory responds to WRITE requests
  - responds to READ, T_READ, RFO or T_RFO when no caches do

Performance Evaluation

- Alternatives
  - Test-and-test-and-set (TTS)
  - Spin locks with exponential backoff
  - MCS software queuing (similar to last class's paper)
  - Hardware queuing: QOSB
    - Add a processor to hardware queue of waiters for a line
    - Allows processor to spin on locally-cached shadow version of line
    - When line is released by processor at head of queue, it is transferred to next waiting processor in queue
  - Load_Linked/Store_Conditional (LL/SC)
    - Load location first (with intent to store)
    - Store a new value only if no updates have occurred to location since load_linked
- Performance in Paper figures 4, 5, 6

Implementation Issues

- Some disadvantages of original technique
  - Uses separate, fully-associative transactional cache
  - Transactional cache size limits transaction length
  - Many implementation details not discussed
- Data version management
  - Need to store old and new data modified by a transaction
  - If using one cache, need to store new data or old data elsewhere
- LogTM
  - Stores new values in cache, old value in per-thread log in virtual memory
  - On commit (common case), log is discarded
  - On abort, old values restored from log by software
- Will TM be successful in making parallel programming easier?

Reading Assignment

- Arthur Veen, "Dataflow Machine Architecture," ACM Computing Surveys, 1986 (Read sections 1, 2, 3 and skim the rest of the paper)
- Gregory Papadopoulos and David Culler, "Monsoon: An Explicit Token-Store Architecture," ISCA, 1990 (Read)