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

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 entries 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:处理器操作（续）

- LT由主动事务发出
  - 如果命中NORMAL条目，则返回其值。
  - 在事务主动中断时，使用XABORT缓存行，或者将XCOMMIT条目分配给此行。
  - 如果T_READ返回BUSY，则主动中断事务（TSTATUS ← false，丢弃所有XABORT条目，将XCOMMIT条目重置为NORMAL）
- LTX由主动事务发出
  - 使用T_RFO在失败时（而不是T_READ）
  - 如果T_RFO成功，则将缓存状态更改为RESERVED
- ST由主动事务发出
  - 类似于LTX，但更新XABORT条目中的数据

Example Implementation: Cache Actions

- 两个常规和事务性缓存都监听总线
  - 忽略所有未在缓存中的地址请求
- 常规缓存操作
  - READ或T_READ：如果状态为VALID，返回值。如果状态为RESERVED或DIRTY，返回值并重置状态为VALID
  - RFO或T_RFO：返回数据并标记自己的行无效
- 事务性缓存操作
  - 如果常规缓存则将TSTATUS设为false或非事务性（READ或RFO）缓存，否则，它忽略带有事务性标签的缓存行
  - T_READ：如果状态为VALID，返回值
  - 所有其他事务性操作：返回BUSY
- 内存响应WRITE请求
  - 响应READ、T_READ、RFO或T_RFO，当没有缓存时

Performance Evaluation

- 可供选择的缓存
  - Test-and-test-and-set (TTS)
  - Spin locks with exponential backoff
  - MCS software queuing (similar to last class’s paper)
  - Hardware queuing: QOSB
    - 在内存队列中添加一个处理器，等待线程
    - 允许处理器在本地缓存中等待
    - 线程被处理后，将之转移到队列中的下一个处理器
  - Load_Linked/Store_Conditional (LL/SC)
    - 加载位置时（以意图存储）
    - 存储新值时，如果自加载以来没有更新

Implementation Issues

- 一些原始技术的缺点
  - 使用单独的全关联事务性缓存
  - 事务性缓存大小限制了事务长度
  - 许多实施细节未讨论
- 数据版本管理
  - 需要存储旧数据，并修改新数据
  - 如果使用一个缓存，需要存储新旧数据
- LogTM
  - 存储新值在缓存中，旧值在虚拟内存
  - 在共享（共同情况），log被丢弃
  - 在主动中断时，旧值被存储
- LogTM是否成功在并行编程中

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)