Computer Science 703

Advance Computer Architecture

2010 Semester 1

Lecture Notes

4May10

Transactional Memory

James Goodman

Department of Computer Science

TEST

- In-class
- Date: Thursday, 6May
- Open notes (no electronic devices)
- Required reading list online: CS703 “Resources”
- Sample test from 2008 (different emphasis) under “Exams”

Notice

NO CLASS ON FRIDAY!

Topics Covered

- Moore’s Law and rise of Multicore
- Multiprocessing, Multithreading & Multicores
- Instruction-Level Parallelism
  - Reducing the cost of branching
  - Branches and branch prediction
  - Dynamic Scheduling (Tomasulo’s algorithm)

Topics Covered (con't)

- Advanced Caching Topics
  - Tuning parameters
    - Size
    - Associativity
    - Line size
  - Compulsory, Capacity, Conflict & Coherence misses
  - Write policies
  - Non-blocking caches
  - Multi-level caches & the inclusion property
  - Victim caches
  - Write buffers

Topics Covered (con't)

- Shared Memory Synchronization
  - Locks
  - Atomic RMW operations
  - Locks and Cache Coherence
- Cache Coherence
  - Snooping & Directories
  - The MOESI model

Topics Covered (con't)

- Methods & Tools
  - Analytical models
  - Simulation
    - Simulating a computer
    - Specifying the model
    - Evaluating the results
    - Validation
  - Benchmarks
  - Performance reporting

Topics Covered (con't)

- Missing Update Problem
- Memory Ordering
  - Litmus tests
  - Sequential Consistency
  - Weak Ordering
  - Processor Consistency
  - Release Consistency

Transactional Memory

- Larus/Rajwar: Transactional Memory (available as PDF through UoA library)
- Herlihy/Moss: Transactional Memory: architectural support for lock-free data structures

Herlihy/Moss 1992

- This paper coined the term “Transactional Memory”
- Really an argument for lock-free data structures to avoid
  - Priority inversion
  - Convoying
  - Deadlock
- Argues for moving responsibility for synchronization away from the programmer
- Points out nice fit with hardware [Knight 1986]

“Experimental evidence suggests that in the absence of inversion, convoying, or deadlock, software implementations of lockfree data structures often do not perform as well as their locking-based counterparts.”

Definition: Transaction

- A transaction is a finite sequence of machine instructions, executed by a single process, satisfying the following properties:
  - Serializability: Transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another. Committed transactions are never observed by different processors to execute in different orders. [includes isolation]
  - Atomicity: Each transaction makes a sequence of tentative changes to shared memory. When the transaction completes, it either commits, making its changes visible to other processes (effectively) instantaneously, or it aborts, causing its changes to be discarded.

New Instructions

- *Load-transactional* (LT) reads the value of a shared memory location into a private register.
- *Load-transactional-exclusive* (LTX) reads the value of a shared memory location into a private register, "hinting" that the location is likely to be updated.
- *Store-transactional* (ST) tentatively writes a value from a private register to a shared memory location. This new value does not become visible to other processors until the transaction successfully commits.

Read set: locations read by LT instructions

Write set: locations read by LTX

New Instructions

"Manipulation" instructions:

- *COMMIT*: make changes permanent (visible). Indicates success/failure
- *ABORT*: discards all updates to the write set
- *VALIDATE*: TRUE indicates transaction has not (yet) aborted

Hardware detecting conflict may cause spontaneous abort

NOTE: no "*Begin Transaction*" operation

Use

Replace critical section with:

1. use LT or LTX to read from a set of locations
2. use VALIDATE to check that the values read are consistent
3. use ST to modify a set of locations,
4. use COMMIT to make the changes permanent.

If either the VALIDATE or the COMMIT fails, the process returns to Step (1).

Basic Idea

"The idea is that the transactional cache holds all the tentative writes, without propagating them to other processors or to main memory unless the transaction commits. If the transaction aborts, the lines holding tentative writes are dropped (invalidated); if the transaction commits, the lines may then be snooped by other processors, written back to memory upon replacement, etc. We assume that since the transactional cache is small and fully associative it is practical to use parallel logic to handle abort or commit in a single cache cycle."

Observations

- No notion of “Begin Transaction”
- What happens when a transaction aborts?
  - No explicit jump/trap on abort
  - Presumably

"The VALIDATE instruction is motivated by considerations of software engineering. A set of values in memory is inconsistent if it could not have been produced by any serial execution of transactions. An orphan is a transaction that continues to execute after it has been aborted (i.e., after another committed transaction has updated its read set). It is impractical to guarantee that every orphan will observe a consistent read set. Although an orphan transaction will never commit, it may be difficult to ensure that an orphan, when confronted with unexpected input, does not store into out-of-range locations, divide by zero, or perform some other illegal action. All values read before a successful VALIDATE are guaranteed to be consistent. Of course, ***VALIDATE is not always needed****, but it simplifies the writing of correct transactions and improves performance by eliminating the need for ad-hoc checks."*

"The implementation described here aborts any transaction that tries to revoke access of a transactional entry from another active transaction. This strategy is attractive if one assumes (as we do) that timer (or other) interrupts will abort a stalled transaction after a fixed duration, so there is no danger of a transaction holding resources for too long.”

From TR: "Deadlock (cyclic waiting) is impossible in this implementation because transactions never wait for one another. A high-priority transaction cannot be delayed indefinitely by a lower-priority transaction, because the latter will be aborted by a timer interrupt if it runs too long. Starvation, however, is still possible. We believe that the best way to avoid starvation is to advise programmers to adopt an adaptive backoff strategy: a transaction that repeatedly aborts should wait for some duration before retrying.

"[F]or programs to be written in a uniform and portable manner, one needs to guarantee at the instruction set architecture level the minimum transaction size that the architecture supports. At present we do not have a good feel for what such a size might be, but it should probably be between 10 and 100. Since one might not want to put a fully associative cache of this size into every implementation of the architecture, schemes that use some hardware but handle larger transactions via software traps seem to be desirable. In fact, one can avoid hard limits on transaction size by offering the software overflow mechanism with all implementations." [TR]

Implementation

[Larus/Rajwar]:

"The *transactional cache is a fully associative cache that holds all transactional writes without propagating their values to other processors or to main memory until the transaction commits. The transactional cache has additional tags with each line that add special meaning to the regular cache states. If tag is* ***empty****, the line has no data. If tag is* ***normal****, the line has committed data. An* ***xcommit*** *tag means the contents must be discarded on commit, and an* ***xabort*** *tag means the contents must be discarded on an abort.*

Implementation

[Larus/Rajwar]:

"The cache coherence protocol is augmented by three new bus cycles. The ***t_read*** *bus cycle is for a transactional read request that goes across the bus. This request can be refused (NACK) by a* ***busy*** *cycle. The* ***t_rfo*** *bus cycle is for a transactional read-for-exclusive request that goes across the bus. This can be refused (NACK) by a busy cycle. The* ***busy*** *bus cycle prevents too many transactions from aborting one another too often. This approach may starve some transactions but a queuing mechanism can address starvation. A* ***busy*** *response does not cause the transaction execution itself to abort immediately but records hardware state to allow the transaction to check for whether the transaction has aborted from the hardware's perspective. Until this check, the transaction may continue to execute without aborting."*