Computer Science 703

Advance Computer Architecture

2006 Semester 1

Lecture Notes

4Apr06

Atomic RMW Memory Operations

James Goodman

Department of Computer Science

Assignments

Tuesday (today):

M. Herlihy and J.E.B. Moss, “Transactional memory: architectural support for lock-free data structures,” *ISCA-20*, pp. 289–300, May 1993.

Wednesday (tomorrow):

R. Rajwar and J. Goodman, "*Speculative lock elision: enabling highly concurrent multithreaded execution*,” *Intl. Symp. on Microarchitecture*, pp. 294-305, Dec 2001.

Friday 14Apr (no class):

Assignment 1 due at 9am.

Atomic Read-Modify-Write Memory Operations

- A third type of memory operation
- Necessary for reasonable implementation of locks and other synchronization mechanisms
- Many variations, not all equivalent
- Must provide mechanism for
  - atomic reading and writing
  - failure detection

A Survey of Primitives

- Atomic Swap
- Test & Set
  - Test & Test & Set
- Fetch&Add (Increment)
  - Combining property
  - This never fails!
- Compare & Swap
  - Scalable!
- Load_Linked/Store_Conditional
- The Oklahoma Update/Transactional Memory

Atomic Swap

Operation: atomically exchange a register value and a memory value

- Might be as little as a single bit
- Test for failure: register indicates bit was already set

Useful for: Acquiring a lock

Reference: ???

Test & Set

Operation: Set memory value (single bit) to 1; report previous value of memory location

- Test for failure: memory bit was already 1
- Variant of Atomic Swap
- Also: Test & Clear

Useful for: Acquiring a lock

Reference: IBM System/360 (1959)

Test & Test & Set

Operation: Two-stage test: don't attempt to set bit until it is clear

- Software implementation: Test + Test&Set
- Test for failure: after second test, same as Test & Set
- No guarantee after first test, but avoids spinning on bus

Useful for: Acquiring a lock, reduced contention

Reference: L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for MIMD parallel processors." In *ISCA-11*, pages 340-347, June 1984.

Fetch&Add

Operation: Atomically add a value to a memory location; set register value to old memory value

- Test for failure: Must be interpreted
- Generalization: Fetch&Φ where Φ is any function that is associative and commutative
- Interesting scalability feature (without cache): combining

Useful for: Simple atomic operations (acquiring a lock, semaphore, assigning unique number)

Reference:

A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, **32**(2), February 1983, pp.175-189.

Compare & Swap

Operation:

- Test if memory location is same as previous value (stored in R1)
- if unchanged, atomically swap memory location and R2
- return success or failure

Powerful primitive: values swapped may be pointers

Reference: IBM System/370 (1970)

MCS Locks

Operation: build software queue using Compare&Swap that allows local spinning and notification when previous lock holder has released lock

Widely used in software today; significant overhead to set up

References:

(1) J.M. Mellor-Crummey & M.L. Scott, "Synchronization without contention," *ASPLOS-4*, pp. 269-278, Apr. 1991 .

(2) T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, **1**(1), p.6-16, January 1990.

Load_Linked/Store_Cond

Operations:

- Load_Linked: Load memory location into R and monitor memory location
- Execute computation
- Store_Conditional: If memory location is known to be undisturbed, write R to memory location
- Return success or failure

Powerful primitive ***in theory***: execute critical section atomically

Reference: E.H. Jensen, G.W. Hagensen, and J.M. Broughton, "A new approach to exclusive data access in shared memory multiprocessors," Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, CA, November 1987.

The Oklahoma Update/ Transactional Memory

Operation: Generalization of LL/SC: multiple memory locations monitored and modified atomically

Powerful primitive ***in theory***: How to implement?

References:

(1) M. Herlihy and J.E.B. Moss, "Transactional memory: architectural support for lock-free data structures," *ISCA-20*, pp. 289–300, May 1993.

(2) J.M. Stone, H.S. Stone, P. Heidelberger, and J. Turek, "Multiple reservations and the Oklahoma update," *IEEE Parallel & Distributed Technology*, **1**(4):58–71, November 1993.

Computer Science 703

Advance Computer Architecture

2006 Semester 1

Lecture Notes

5Apr06

Transactional Memory

James Goodman

Department of Computer Science

What's Wrong with Locks?

- ***Priority*** inversion occurs when a lower-priority process is preempted while holding a lock needed by higher-priority processes.
- ***Convoying*** occurs when a process holding a lock is rescheduled, perhaps by exhausting its scheduling quantum, by a page fault, or by some other kindl of interrupt. When such an interruption occurs, other processes capable of running maybe unable to progress.
- ***Deadlock*** can occur if processes attempt to lock the same set of objects in different orders, Deadlock avoidance can be awkward if processes must lock multiple data objects, particularly if the set of objects is not known in advance.

Transactional Memory

Basic insight behind Transactional Memory:

- Can generalize LL/SC to handle multiple reads and writes
- Invalidation-based cache coherence protocols can be used to detect transaction conflicts.
- By using the existing cache coherence protocol, atomic transactions can be supported cheaply.

Reference: M. Herlihy and J.E.B. Moss, "Transactional memory: architectural support for lock-free data structures," *ISCA-20*, pp. 289–300, May 1993.

Lock-Free & Wait-Free Algorithms

In contrast to algorithms that protect access to shared data with locks, lock-free and wait-free algorithms are specially designed to allow multiple threads to read and write shared data concurrently without corrupting it.

Lock-free refers to the fact that a thread cannot lock up: every step it takes brings progress to the system. This means that no synchronization primitives such as *mutexes* or *semaphores* can be involved, as a lock-holding thread can prevent global progress if it is switched out.

Wait-free refers to the fact that a thread can complete any operation in a finite number of steps, regardless of the actions of other threads. It is possible for an algorithm to be lock-free but not wait-free.

Source: Wikipedia (http://en.wikipedia.org/wiki/Lock-free_and_wait-free_algorithms)

Nonblocking, Lock-free, Wait-Free

- ***Nonblocking*** algorithms guarantee that if there are one or more active processes trying to perform operations on a shared data structure, *some* operation will complete within a finite number of time steps.
- A ***wait-free*** algorithm is both non-blocking and starvation free: it guarantees that every active process will make progress within a bounded number of time steps.
- A ***lock-free*** algorithm may not be non-blocking, i.e., it does not use locking mechanisms, but allows a slow process to delay faster processes indefinitely.

Reference: M.M. Michael and M.L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,"*15th ACM Symp. on Principles of Distributed Computing*, May 1996.

The Critical Section

```
CriticalSection() {
    acquire(lock);
    read(data1);
    read(data2);
     …
    write(data1);
     …
    release(lock);
}
```

```
acquire(lock) {
    while (swap(lock,HELD) != FREE)
            ;
     MemBar();
}

release(lock) {
     MemBar();
     lock = FREE;
}
```


Transactions

```
atomic {
  mumble;
} catch (AbortedException e) {
}
```

Programmer defines transaction scope

Problems with Transactional Memory

- Requires six new instructions for programmers to use
- Uses an extra cache called the transactional cache to buffer optimistic updates
- Supports arbitrary read-modify-write operations, size of the operations limited only by the processor's transactional cache.
- Requires programmers to reason about correctness of lock-free algorithms.

Speculative Lock Elision

(Class presentation, no notes)

Computer Science 703

Advance Computer Architecture

2006 Semester 1

Lecture Notes

6Apr06

HW & SW Transactional Memory

James Goodman

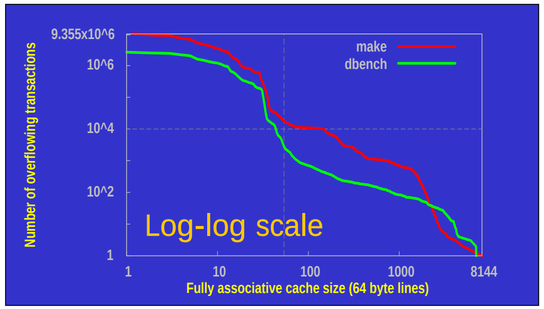
Department
of
Computer Science

SLE: the Ultimate Solution?

Good start, but not the end

1. What to do about conflicts?
   - Transactional Lock Removal (TLR)
     - Use SLE, but resolve conflicts in hardware, queueing requests for data
     - Must deal with deadlock problem: two cache lines requested in different order
2. What about long-running transactions?
   - Speculation requires duplicating state
   - Cache will eventually overflow
   - How to handle transactions that don't fit in the cache?

Transactional Linux



- Almost all of the transactions require < 100 cache lines
  - 99.9% need fewer than 54 cache lines
- There are, however, some very large transactions!
  - >500k-byte fully-associative cache required

Source: Unbounded Transactional Memory, MIT

Virtual Transactional Memory (VTM)

- Assumes high-speed scheme for common case (SLE)
- Only an overflow mechanism
  - No overhead on common in-cache case
    - Check shared overflow counter on cache miss
  - Low overhead when no conflict
    - Shared Bloom Filter rules out conflicts
    - Filter resides in virtual memory
  - Higher overhead on possible conflict
    - Hardware table walk to detect actual conflict
    - Table resides in virtual memory
    - Only incurred by large transactions with likely conflict
- Supports context switches and paging

R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," *ISCA-32*, Jun. 2005.

VTM Structures

- XSW--Transaction Status Word register
  - Running
  - Aborted
  - Committing (updates not yet visible
- XADT: Transaction Address Data Table
  - Common to all transactions sharing the address space
  - Table of overflowed cache lines

Making Common Case Fast

- On cache miss
- Check overflow flag
- If overflow, check XADT
  - Special access to XADT: Bloom filter
    - High-speed check with possible false positives
  - If positive, walk XADT table to find element
- On commit, if overflow, make XADT entries visible
- While committing, conflict detected in overflowed cannot return old value.
  - Other accesses to XADT may be delayed during this (rare) phase

LogTM: Log-based Transactional Memory

Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill & David A. Wood

Multifacet Project (www.cs.wisc.edu/multifacet)
Directed by Mark D. Hill & David A. Wood
Computer Sciences Department
University of Wisconsin—Madison

Appeared in HPCA 2006

LogTM Summary

- Chip multiprocessors make threaded programming important
- But locks challenging (to get simplicity & performance)
- Transactional Memory (TM) promising
- begin_transaction { atomic execution } end_transaction
- Existing (Hardware) TMs
  - Mostly keep Old values "in place" & New values "elsewhere"
  - Commits slower than aborts, but commits more common
- New LogTM: Log-based Transactional Memory
  - Old values to log in thread-private virtual memory (like DBMSs)
  - New values "in place" to make common commits fast
  - Also allows cache overflow & software abort handling
- See http://www.cs.wisc.edu/multifacet/

Software Transactional Memory

- Transactions can be handled entirely in software
- So far, implementations are very slow
  - Hardware is fundamentally parallel
  - Software is fundamentally serial, not easy to parallelize

Dynamic Software Transactional Memory (DSTM)

M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III, "Software Transactional Memory for Dynamic-Sized Data Structures," *Twenty-Second ACM Symp. on Principles of Distributed Computing (PODC), Boston, Massachusetts*, Jul. 2003.

State of the Art

- Hardware is fast, but has hard resource limits
- Software has vastly greater hardware limits, but is slow

Hybrid?