Consistency & TM

Today's topics:

Consistency models
  the “when” of the CC-NUMA game
Transaction Memory
  an alternative to lock based synchronization
  additional reading: paper from HPCA 2006
  on class web page

Consistency

• For DSM systems
  • cache coherence
    » ensure multiple processors see a consistent memory view
    » does not answer “how consistent”
    * e.g. when are things truly consistent
  • consistency models
    » there are several ... but first a little detour

• Programs share variables
  • problem redefined
    » when does a write in some processor become visible to a read
      in another processor?
    » OR what properties must be enforced between reads and
      writes on different processors?
Consider

- **Code sequences on 2 different processors**
  
  P1: A= 0; (cycle 1)  
  A=1; (cycle n)  
  if (B==0) … (cycle n+1)
  
  P2: B=0; (cycle 1)  
  B=1; (cycle n)  
  if (A==0) … (cycle n+1)

- **If A and B are cached on both processors**
  
  - **problem – inherent race**
    - P1 can read 2 possible values written by P2
    - and vice versa
    - what if the correct last write invalidate from P2 is not seen by P1 before the read?
  
  - **non-deterministic program**
  
  - **Question is whether or not this behavior should be allowed?**
    - and if so under what conditions?

---

Enter Consistency Models

- **Sequential consistency**
  
  - **result of any execution must be the same**
    - if memory accesses by each processor are kept in order
    - and the accesses of all processors are arbitrarily interleaved
      - note in previous code segments this won't be the case
        - only way out here is for the programmer to synchronize the order
        - locks
  
  - **simplest implementation**
    - sequence ALL memory transactions
  
  - for DSM this means
    - synchronizing all memory transactions at a global atomicity point
    - intractable due to performance impediments
  
  - **fence instructions (slightly better)**
    - system wide flush of all pending memory references
  
  - **pro's and con's**
    - + programmer sees a simple deterministic model
      - complicated - same as hazard solutions for pipelines Wax and RAW
    - - slow – hard to swallow in a parallel world
Program Synchronization

- **Programmer must specify order that matters**
  - locks, barriers, whatever → data race free behavior
  - other non-determinacy is accepted
    - exception concurrent writer problem
      - such as CC-NUMA/DSM write-invalidate protocol
- **obvious problems**
  - additional complexity pushed onto the programmer
    - more heinous for fine-grain locks
  - synchronization = serialization
    - defeats performance advantage of parallelism

- **Relaxing consistency**
  - hardware allows some/most memory operations to happen out of order
    - several variants
  - programmer still has to control orderings that matter
    - critical sections, locks, ...

Book Terminology

- **Attempt to be compatible with your textbook**
- **X → Y**
  - implies X must happen before Y
  - candidate values for X & Y are READS and WRITES (R,W)
  - hence options
    - R → R
    - R → W
    - W → R
    - W → W

- **Relaxing the R → R constraint**
  - this is essentially sequential consistency
    - although your book doesn't view it this way
  - think about it
    - reordering reads doesn't change RAW or Wax hazards
    - the problem happens when you promote either reads or writes over a write!!
Relaxed Consistency Models

- **TSO – total store ordering**
  - relax W→R
  - retain write ordering but allow reads to be reordered
    - note this changes RAW hazard behavior
    - fix: programmer synchronization
  - benefit
    - write buffering works
    - lots of programs just want the latest value
  - a.k.a. “processor consistency”
    - from a single processor view point
    - reordering reads doesn’t change anything

- **PSO – partial store order**
  - relax W→W ordering (WAW hazards don’t matter)
    - note this does NOT mean to the same location
    - requires total memory disambiguation
      - easy at main memory where physical addresses are used
      - independent writes can be reordered

Relaxed Consistency II

- **Weak ordering & release consistency**
  - relaxing R→W and R→R ordering
    - meaning – don’t care about WAR or RAW hazards
  - reality
    - if threads rarely interact
      - reduced synchronization → improved parallelism
      - memory system can respond out of order
      - big performance gain

- **Problem w/ relaxed consistency**
  - programmer needs to know what the compiler/hardware supports
    - may be able to specify the consistency model
  - bottom line
    - programmer needs to explicitly synchronize the things that matter
      - in a concurrent world this can’t be avoided anyway
What's the Point?

• Modes in today's hardware
  • allow various consistency models
  • more relaxed is potentially faster
    » but programmer needs to know what to explicitly synchronize
    » and this depends on the mode
  • vocabulary
    » changes a bit by vendor
    » previous terminology is the most common

Enter TM

• Transactional Memory
  • original idea from TK@MIT
    » take a data base idea and apply it to the shared memory problem
  • note that there are lots of variants
    » idea today is a shallow dive into the space
  • basic idea
    » program: transaction consists of an atomic block
      • read stuff
      • do something
      • write stuff
    » if nothing else interacts w/ stuff then all is well
      • otherwise abort and don't do anything destructive
        - e.g. write to memory
    » similar to svn
      • e.g. version management but with all or nothing success idea
What Changes?

• New TM model simplifies programming
  • lock-unlock replaces the transaction begin-end
  • programmer specifies the code sections that are viewed as atomic
    » hardware and/or system software insures this atomicity illusion
    » even though conflicts may occur
  • adv: eliminates deadlock
    » lock-unlock is blocking
      • mutually dependent locks incrementally obtained?
        - 5 dining philosopher problem – deadlock possible
        - key incremental claiming with no “give back”
    » transactions are non-blocking
      • go ahead and conflict but abort if you do then abort
        – effectively a “give back”
  • disadv: if lots of aborts (stats show this is rare)
    » then lots of activity for little progress
      • power wasted for no productive reason

TM Basics

• Must provide
  • atomicity
    » transaction succeeds or fails – all or nothing
    » no partial success allowed
  • isolation
    » intermediate transaction state invisible to other transactions to the same or overlapping data space
  • version management
    » keep track of which version is correct “latest”
      • handles simultaneous storage of new data – visible on commit
      • and old data which is retained if transaction aborts
  • conflict detection
    » mechanism to determine whether interleaving transactions are happening
    » essentially a transaction consistency detector
      • signals overlap of this transaction write set with other transaction read or write set
Conflict Detection & Version Mgmt.

- Key to the game
- 2 variants for version management
  - eager – put the new value in place
    » restore old version on abort
  - lazy – leave old value
    » replace with new values on commit
    » new values can be cached
- 2 variants of conflict management
  - eager – detect offending loads or stores immediately
  - lazy – defer detection until commit time

Previous Work

- TCC – trans. coherence & consistency - Hammond et al.
  - lazy on both
    » very similar to DBMS approach using optimistic concurrency control (OCC)
    » stores new in L1 – overwrites in L2 @ commit time
    » detect conflict when other transactions commit
- LTM – Large TM – Ananian et al.
  - lazy version mgmt, eager conflict detection
    » old value in main memory, new values are cached
      » coherence protocol stores 2 different values at same address
      - e.g. delayed consistency with main mem
      » write miss stores to main mem hash table
    » conflict detection – invalidation of write set
      » complicated by overflow hash table
      » SW has to walk before knowing – potential problem
Previous Work II

- **VTM – virtual TM – Rajwar et al.**
  - lazy version mgmt. & eager conflict mgmt.
- **UTM – unbounded TM – Ananian et al. (again)**
  - eager – eager
    - uses conservative concurrency control (CCC)
    - problem = complex
      - pointer per memory block
      - linked list log of both reads and writes
- **Ideal**
  - eager – eager but without the UTM overhead
  - if the common case is that conflicts are rare
    - then eager – eager overhead is reduced since aborts are rare
    - Amdahl's law appears again

---

LogTM: Log-based Transactional Memory

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

Slide Credit: following slides from Kevin Moore’s presentation at HPCA06
(slightly edited)
Big Picture

• (Hardware) Transactional Memory promising
  • Most use lazy version management
    » Old values “in place”
    » New values “elsewhere”
  • Commits slower than aborts

• New LogTM: Log-based Transactional Memory
  • Uses eager version management (like most databases)
    » Old values to log in thread-private virtual memory
    » New values “in place”
  • Makes common commits fast!
  • Allows cache overflow
  • Aborts handled in software

LogTM’s Eager Version Management

• Old values stored in the transaction log
  • A per-thread linear (virtual) address space (like the stack)
  • Filled by hardware (during transactions)
  • Read by software (on abort)

• New values stored “in place”
• Current design requires hardware support
**Transaction Log Example**

- **Initial State**
  - LogBase = LogPointer
  - TM count > 0

---

**Data Block**

<table>
<thead>
<tr>
<th>VA</th>
<th>00</th>
<th>40</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>23--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>41--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**Log Base**

- 1000

**Log Ptr**

- 1000

**TM count**

- 1

---

**Transaction Log Example**

- **Store r2, (c0) /* r2 = 56 */**
  - Set W bit for block (c0)
  - Store address (c0) and old data on the log
  - Increment Log Ptr to 1048
  - Update memory

---

**Data Block**

<table>
<thead>
<tr>
<th>VA</th>
<th>00</th>
<th>40</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>23--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>41--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**Log Base**

- 1000

**Log Ptr**

- 1048

**TM count**

- 1
**Transaction Log Example**

**Commit transaction**
- Clear R & W for all blocks
- Reset Log Ptr to Log Base (1000)
- Clear TM count

<table>
<thead>
<tr>
<th>VA</th>
<th>Data Block</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Abort transaction**
- Replay log entries to "undo" the transaction
- Reset Log Ptr to Log Base (1000)
- Clear R & W bits for all blocks
- Clear TM count

<table>
<thead>
<tr>
<th>VA</th>
<th>Data Block</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eager Version Management Discussion

- **Advantages:**
  - Fast Commits
    - No copying
    - Common case
  - Unambiguous Data Values
    - Value of a memory location is the value of the last store (no table lookup)

- **Disadvantages**
  - Slow/Complex aborts
    - Undo aborting transaction
  - Relies on Eager Conflict Detection/Prevention

LogTM’s Eager Conflict Detection

- **Most Hardware TM Leverage Invalidation Cache Coherence**
  - Add per-processor transactional write (W) & read (R) bits
  - Coherence protocol detects transactional data conflicts
  - E.g., Writer seeks M copy, seeks S copies, & finds R bit set

- **LogTM detects conflicts this way using directory coherence**
  - Requesting processor issues coherence request to directory
  - Directory forwards to other processor(s)
  - Responding processor detects conflict using local R/W bits & informs requesting processor of conflict
MOESI Protocol

- Extension of MESI protocol (local line state)
  - M – “modified”
    » exclusive dirty, main memory not consistent
  - O – “owned”
    » only 1 owner but other sharers can exist
    » may or may not be dirty
    » main mem may be inconsistent
    » adv: deferred update of main memory
  - E – “exclusive clean”
    » main mem consistent
  - S – “shared clean”
    » main mem consistent
  - I – “invalid”

- Directory global state change
  - new/old: indicates whether main memory is consistent
  - what others states?

Write Miss Example

- P0 store
  - P0 sends get exclusive (GETX) request
  - Directory responds with data (old marked)
  - P0 executes store
    » sets W

Diagram:
- P0
  - TM mode: 1
  - Overflow: 0
  - State: M (-W) [new]
- P1
  - TM mode: 0
  - Overflow: 0
  - State: I (-- [none]

Directory
- M@P0 [old]
Get Shared w/ Existing M copy

- **In-cache transaction conflict**
  - P1 sends get shared (GETS) request
  - Directory forwards to P0
  - P1 detects conflict (M state) and sends NACK

Cache Overflow Victim

- **Cache overflow**
  - P0 sends put exclusive (PUTX) request
  - Directory acknowledges
  - P0 sets overflow bit
  - P0 writes data back to memory
Remote GETS after Overflow

- Out-of-cache conflict
  - P1 sends GETS request
  - Directory forwards to P0
  - P0 detects a (possible) conflict
  - P0 sends NACK

Commit

- Quick commit
  - P0 clears TM mode and Overflow bits
  - Everything updated already
Deferred Clean

- **Lazy cleanup**
  - P1 sends GETS request
  - Directory forwards request to P0
  - P0 detects no conflict, sends CLEAN
  - Directory sends Data to P1

LogTM's Conflict Detection w/ Cache Overflow

- **At overflow at processor P**
  - Set P's overflow bit (1 bit per processor)
  - Allow writeback, but set directory state to Sticky@P

- **At transaction end (commit or abort) at processor P**
  - Reset P's overflow bit

- **At (potential) conflicting request by processor R**
  - Directory forwards R's request to P.
  - P tells R "no conflict" if overflow is reset
  - But asserts conflict if set (w/ small chance of false positive)
Conflict Resolution

• Conflict Resolution
  • Can wait risking deadlock
  • Can abort risking livelock
  • Wait/abort transaction at requesting or responding proc?

• LogTM resolves conflicts at requesting processor
  • Requesting processor waits (using coherence nacks/retries)
  • But aborts if other processor is waiting (deadlock possible) & it is logically younger (using timestamps)

• Future: Requesting processor traps to software contention manager that decides who waits/aborts

---

Evaluation

• Simulated Machine: 32-way non-CMP
  • 32 SPARC V9 processors running Solaris 9 OS
  • 1 GHz in-order processors w/ ideal IPC=1 & private caches
  • 16 kB 4-way split L1 cache, 1 cycle latency
  • 4 MB 4-way unified L2 cache, 12 cycle latency
  • 4 GB main memory, 80-cycle access latency
  • Full-bit vector directory w/ directory cache

• Simulation Infrastructure
  • Virtutech Simics for full-system function
  • Magic no-ops instructions for begin_transaction() etc.
  • Multifacet GEMS for memory system timing (Ruby only)
    GPL Release: http://www.cs.wisc.edu/gems/
  • LogTM simulator part of GEMS 2.0 (coming soon)
Microbenchmark Analysis

- **Shared Counter**
  - All threads update the same counter
  - High contention
  - Small Transactions

- **LogTM v. Locks**
  - EXP - Test-And-Test-And-Set Locks with Exponential Backoff
  - MCS - Software Queue-Based Locks

BEGIN_TRANSACTION();

new_total = total.count + 1;
private_data[id].count++;
total.count = new_total;

COMMIT_TRANSACTION();

- LogTM (like other HTMs) does not read/write lock
- LogTM has few aborts despite conflicts
SPLASH-2 Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>512 Bodies</td>
<td>Locks on tree nodes</td>
</tr>
<tr>
<td>Cholesky</td>
<td>14</td>
<td>Task queue locks</td>
</tr>
<tr>
<td>Ocean</td>
<td>Contiguous partitions, 258</td>
<td>Barriers</td>
</tr>
<tr>
<td>Radiosity</td>
<td>Room</td>
<td>Task queue and buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>locks</td>
</tr>
<tr>
<td>Raytrace</td>
<td>Small image (teapot)</td>
<td>Work list and counter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>locks</td>
</tr>
<tr>
<td>Raytrace-Opt</td>
<td>Small image (teapot)</td>
<td>Work list and counter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>False sharing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>optimization</td>
</tr>
<tr>
<td>Water N-Squared</td>
<td>512 Molecules</td>
<td>False sharing optimization</td>
</tr>
</tbody>
</table>

SPLASH2 Benchmark Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% Stalls</th>
<th>% Aborts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>4.89</td>
<td>15.3</td>
</tr>
<tr>
<td>Cholesky</td>
<td>4.54</td>
<td>2.07</td>
</tr>
<tr>
<td>Ocean</td>
<td>.30</td>
<td>.52</td>
</tr>
<tr>
<td>Radiosity</td>
<td>3.96</td>
<td>1.03</td>
</tr>
<tr>
<td>Raytrace-Base</td>
<td>24.7</td>
<td>1.24</td>
</tr>
<tr>
<td>Raytrace-Opt</td>
<td>2.04</td>
<td>.41</td>
</tr>
<tr>
<td>Water</td>
<td>0</td>
<td>.11</td>
</tr>
</tbody>
</table>
Conclusion

• Commits far more common than aborts
  • Conflicts are rare
  • Most conflicts can be resolved w/o aborts
  • Software aborts do not impact performance
• Overflows are rare (in current benchmarks)
• LogTM
  • Eager Version Management makes the common case (commit) fast
  • Sticky States/Lazy Cleanup detects conflicts outside the cache (if overflows are infrequent)
  • More work is needed to support virtualization and OS interaction
• False sharing has greater impact with TM

Concluding Remarks

• Highly concurrent machines are here
  • trend is clear
• Future programming model
  • unclear
    » message passing – MPI, OpenMPI, MCAPI
    » some leverage from shared memory
    • like LogTM
    » other transactional memory models
• Personal opinion
  • better HW support for light overhead MP will win
  • some coherent shared memory on socket likely
    » inflection point is unknown