EECS 570
Lecture 8
Transactional Memory
Winter 2016
Prof. Thomas Wenisch

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinhardt, Roth, Smith, Singh, and Wenisch. Special acknowledgement to M. Martin for Transactional Memory slides.
Announcements

Project proposals due tomorrow

• No office hours or discussion Fri (project meetings)
  - Sign up for a meeting time by replying on Piazza
  - First group to reply gets to claim slot
Readings

For Today:

For next Monday:
- Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence (Ch. 6 & 7)
Ticket Locks

• To ensure fairness and reduce coherence storms

• Locks have two counters: next_ticket, now_serving
  □ Deli counter

• acquire(lock_ptr):
  □ my_ticket = fetch_and_increment(lock_ptr->next_ticket)
  □ while(lock_ptr->now_serving != my_ticket); // spin

• release(lock_ptr):
  □ lock_ptr->now_serving = lock_ptr->now_serving + 1
    □ (Just a normal store, not an atomic operation, why?)

• Summary of operation
  □ To “get in line” to acquire the lock, CAS on next_ticket
  □ Spin on now_serving
Ticket Locks

• Properties
  ❑ Less of a “thundering herd” coherence storm problem
    ❑ To acquire, only need to read new value of now_serving
  ❑ No CAS on critical path of lock handoff
    ❑ Just a non-atomic store
  ❑ FIFO order (fair)
    ❑ Good, but only if the O.S. hasn’t swapped out any threads!

• Padding
  ❑ Allocate now_serving and next_ticket on different cache blocks
    ❑ struct { int now_serving; char pad[60]; int next_ticket; } ...
  ❑ Two locations reduces interference

• Proportional backoff
  ❑ Estimate of wait time: (my_ticket - now_serving) * average hold time
Array-Based Queue Locks

• Why not give each waiter its own location to spin on?
  - Avoid coherence storms altogether!

• Idea: “slot” array of size N: “go ahead” or “must wait”
  - Initialize first slot to “go ahead”, all others to “must wait”
  - Padded one slot per cache block,
    - Keep a “next slot” counter (similar to “next_ticket” counter)

• Acquire: “get in line”
  - my_slot = (atomic increment of “next slot” counter) mod N
  - Spin while slots[my_slot] contains “must_wait”
  - Reset slots[my_slot] to “must wait”

• Release: “unblock next in line”
  - Set slots[my_slot+1 mod N] to “go ahead”
Array-Based Queue Locks

• Variants: Anderson 1990, Graunke and Thakkar 1990

• Desirable properties
  □ Threads spin on dedicated location
    ○ Just two coherence misses per handoff
    ○ Traffic independent of number of waiters
  □ FIFO & fair (same as ticket lock)

• Undesirable properties
  □ Higher uncontended overhead than a TTS lock
  □ Storage $O(N)$ for each lock
    ○ 128 threads at 64B padding: 8KBs per lock!
    ○ What if $N$ isn’t known at start?

• List-based locks address the $O(N)$ storage problem
  □ Several variants of list-based locks: MCS 1991, CLH 1993/1994
List-Based Queue Lock (MCS)

- A “lock” is a pointer to a linked list node
  - next node pointer
  - boolean must_wait
  - Each thread has its own local pointer to a node “I”

- acquire(lock):
  I->next = null;
  predecessor = fetch_and_store(lock,I)
  if predecessor != nil //some node holds lock
      I->must_wait = true
      predecessor->next = I //predecessor must wake us
  repeat while I->must_wait //spin till lock is free

- release(lock):
  if (I->next == null) //no known successor
      if compare_and_swap(lock,I,nil) //make sure...
          return //CAS succeeded; lock freed
      repeat while I->next = nil //spin to learn successor
      I->next->must_wait = false //wake successor
MCS Lock Example: Time 0

- acquire(lock):
  I->next = null;
  pred = FAS(lock, I)
  if pred != nil
    I->must_wait = true
  pred->next = I
  repeat while I->must_wait

- release(lock):
  if (I->next == null)
    if CAS(lock, I, nil)
      return
    repeat while I->next == nil
  I->next->must_wait = false

· acquire(lock):
  I->next = null;
  pred = FAS(lock, I)
  if pred != nil
    I->must_wait = true
  pred->next = I
  repeat while I->must_wait

· release(lock):
  if (I->next == null)
    if CAS(lock, I, nil)
      return
    repeat while I->next == nil
  I->next->must_wait = false
MCS Lock Example: Time 1

- \( t_1 \): Acquire(L)

\[\begin{align*}
\text{acquire(lock)}: & \\
& \text{I->next} = \text{null}; \\
& \text{pred} = \text{FAS(lock, I)} \\
& \text{if pred} \neq \text{nil} \\
& \text{I->must\_wait} = \text{true} \\
& \text{pred->next} = \text{I} \\
& \text{repeat while I->must\_wait}
\end{align*}\]

\[\begin{align*}
\text{release(lock)}: & \\
& \text{if (I->next == null)} \\
& \quad \text{if CAS(lock, I, nil)} \\
& \quad \text{return} \\
& \quad \text{repeat while I->next == nil} \\
& \text{I->next->must\_wait} = \text{false}
\end{align*}\]
MCS Lock Example: Time 2

- \( t_1 \): Acquire(L)
- \( t_2 \): Acquire(L)

- **acquire(lock):**
  - \( I->next = null; \)
  - pred = FAS(lock,I)
  - if pred != nil
    - \( I->must\_wait = true \)
    - pred->next = I
    - repeat while I->must_wait

- **release(lock):**
  - if (I->next == null)
    - if CAS(lock,I,nil)
      - return
    - repeat while I->next == nil
  - I->next->must\_wait = false
MCS Lock Example: Time 3

- $t_1$: Acquire(L)
- $t_2$: Acquire(L)
- $t_3$: Acquire(L)

```
• acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
      I->must_wait = true
      pred->next = I
  repeat while I->must_wait

• release(lock):
  if (I->next == null)
      if CAS(lock,I,nil)
          return
  repeat while I->next == nil
  I->next->must_wait = false
```
MCS Lock Example: Time 4

- $t_1$: Acquire($L$)
- $t_2$: Acquire($L$)
- $t_3$: Acquire($L$)
- $t_1$: Release($L$)

- **acquire(lock):**
  
  ```
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
    I->must_wait = true
  pred->next = I
  repeat while I->must_wait
  ```

- **release(lock):**
  
  ```
  if (I->next == null)
    if CAS(lock,I,nil)
      return
  repeat while I->next == nil
  I->next->must_wait = false
  ```
MCS Lock Example: Time 5

- $t_1$: Acquire($L$)
- $t_2$: Acquire($L$)
- $t_3$: Acquire($L$)
- $t_1$: Release($L$)
- $t_2$: Release($L$)

- acquire($lock$):
  - $I->next = null$;
  - $pred = FAS(lock,I)$
  - if $pred != nil$
    - $I->must_wait = true$
    - $pred->next = I$
  - repeat while $I->must_wait$

- release($lock$):
  - if ($I->next == null$)
    - if CAS($lock,I,nil$)
      - return
    - repeat while $I->next == nil$
  - $I->next->must_wait = false$
MCS Lock Example: Time 6

- $t_1$: Acquire(L)
- $t_2$: Acquire(L)
- $t_3$: Acquire(L)
- $t_1$: Release(L)
- $t_2$: Release(L)
- $t_3$: Release(L)

 acquisition:

\[
\text{acquire}(\text{lock}): \\
\text{I->next} = \text{null}; \\
\text{pred} = \text{FAS(lock, I)} \\
\text{if pred} \neq \text{nil} \\
\text{I->must\_wait} = \text{true} \\
\text{pred->next} = \text{I} \\
\text{repeat while I->must\_wait}
\]

 release:

\[
\text{release}(\text{lock}): \\
\text{if (I->next == null)} \\
\text{if CAS(lock, I, nil)} \\
\text{return} \\
\text{repeat while I->next == nil} \\
\text{I->next->must\_wait} = \text{false}
\]

**release() w/o CAS is more complex; see paper**
Queue-based locks in HW: QOLB

- **Queue On Lock Bit**
  - HW maintains doubly-linked list between requesters
    - This is a key idea of “Scalable Coherence Interface”, see Unit 3
  - Augment cache with “locked” bit
    - Waiting caches spin on local “locked” cache line
  - Upon release, lock holder sends line to 1st requester
    - Only requires one message on interconnect
Fundamental Mechanisms to Reduce Overheads
[Kägi, Burger, Goodman ASPLOS 97]

- **Basic mechanisms**
  - Local Spinning
  - Queue-based locking
  - Collocation
  - Synchronous Prefetch

<table>
<thead>
<tr>
<th></th>
<th>Local Spin</th>
<th>Queue</th>
<th>Collocation</th>
<th>Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp;S</td>
<td>No</td>
<td>No</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>T&amp;T&amp;S</td>
<td>Yes</td>
<td>No</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>MCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>QOLB</td>
<td>yes</td>
<td>Yes</td>
<td>Optional</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Microbenchmark Analysis

![Graph showing relative sync period against the number of CPUs. The graphs for T&S, T&T&S, MCS, and QOLB are compared. The graph includes data from Kägi 97.]
Performance of Locks

• Contention vs. No Contention
  □ Test-and-Set best when no contention
  □ Queue-based is best with medium contention
  □ Idea: switch implementation based on lock behavior
    ○ Reactive Synchronization – Lim & Agarwal 1994
    ○ SmartLocks – Eastep et al 2009

• High-contention indicates poorly written program
  □ Need better algorithm or data structures
Point-to-Point Event Synchronization

- Can use normal variables as flags
  
  ```
  a = f(x);
  flag = 1;
  ```

  ```
  while (flag == 0);
  b = g(a);
  ```

- If we know initial conditions
  
  ```
  a = f(x);
  ```

  ```
  while (a == 0);
  b = g(a);
  ```

- **Assumes Sequential Consistency!**

- Full/Empty Bits
  
  - Set on write
  
  - Cleared on read
  
  - Can’t write if set, can’t read if clear
Barriers
Barriers

• Physics simulation computation
  □ Divide up each timestep computation into N independent pieces
  □ Each timestep: compute independently, synchronize

• Example: each thread executes:

  segment_size = total_particles / number_of_threads
  my_start_particle = thread_id * segment_size
  my_end_particle = my_start_particle + segment_size - 1
  for (timestep = 0; timestep += delta; timestep < stop_time):
    calculate_forces(t, my_start_particle, my_end_particle)
    barrier()
    update_locations(t, my_start_particle, my_end_particle)
    barrier()

• Barrier? All threads wait until all threads have reached it
Example: Barrier-Based Merge Sort
Global Synchronization Barrier

- At a barrier
  - All threads wait until all other threads have reached it

- Strawman implementation (wrong!)

```plaintext
global (shared) count : integer := P

procedure central_barrier
  if fetch_and_decrement(&count) == 1
    count := P
  else
    repeat until count == P

- What is wrong with the above code?
Sense-Reversing Barrier

• Correct barrier implementation:

```plaintext
global (shared) count : integer := P
global (shared) sense : Boolean := true
local (private) local_sense : Boolean := true

procedure central_barrier
   // each processor toggles its own sense
   local_sense := !local_sense
   if fetch_and_decrement(&count) == 1
      count := P
      // last processor toggles global sense
      sense := local_sense
   else
      repeat until sense == local_sense

• Single counter makes this a “centralized” barrier
```
Other Barrier Implementations

- Problem with centralized barrier
  - All processors must increment each counter
  - Each read/modify/write is a serialized coherence action
    - Each one is a cache miss
  - $O(n)$ if threads arrive simultaneously, slow for lots of processors

- Combining Tree Barrier
  - Build a $\log_k(n)$ height tree of counters (one per cache block)
  - Each thread coordinates with $k$ other threads (by thread id)
  - Last of the $k$ processors, coordinates with next higher node in tree
  - As many coordination address are used, misses are not serialized
  - $O(\log n)$ in best case

- Static and more dynamic variants
  - Tree-based arrival, tree-based or centralized release
Transactional Memory

Thanks to M.M.K. Martin of U. Penn for many of these slides
Motivational Challenge Problem

- A concurrent “set” data structure that supports:
  - insert(Set s, key k)
  - lookup(Set s, key k)
  - delete(Set s, key k)

- Ok, now extend it to add:
  - transfer(Set s1, Set s2, key k)
  - Key k must always be in one set (never both or neither)

- Even with coarse-grained locking...
  - Breaks abstraction: exposes internal lock
  - Deadlock concern: which set’s lock to grab first?
“Ideal” Solution to Challenge

- How to transfer a key between two sets?
  ```c
  void transfer(Set s1, Set s2, key k) {
    atomic {
      delete(s1, k);
      insert(s2, k);
    }
  }
  ```

- Where “atomic” has:
  - Simplicity of coarse-grained locking
  - Concurrency of fine-grained locking
  - Without fine-grain locking overheads

  The promise of “transactional memory”
Transactional Memory: The Next Big Thing™

- Region that executes serially (isolated/atomic)
  - Inspired by database transactions, *but different*

- Implementation: *speculative execution*
  - Serialize only on dynamic conflicts (eager or lazy)
    - e.g., when key manipulated by different threads
  - Partly overcomes the granularity/complexity tradeoff
    - Avoid conservative serialization of locking
Hot, Hot, Hot!

- Pioneering work
  - HTM [Herlihy+, ISCA’93], Oklahoma Update [Stone+, ‘93]
    -- years pass ---

- Speculative locking
  - E.g., SLE/TLR [Rajwar+, MICRO ‘01 & ASPLOS ‘02]

- Software Transactional Memory
  - E.g., DSTM [Herlihy+, PODC ‘03], [Harris+, OOPSLA ‘03], more

- Hardware Transactional Memory
  - E.g., TCC [Hammond+, ISCA ‘04 & ASPLOS ‘04],
    UTM [Ananian+, HPCA ‘05], VTM [Rajwar+, ISCA ‘05]
    LogTM [Moore+, HPCA ‘06], and more...

- Hardware/software hybrids...

  Lots of TM papers in recent years

  300+ citations in “Transactional Memory”, 2nd Edition, 2010
Garner's Hype Cycle

- Peak of Inflated Expectations (~2006)
- Plateau of Productivity
- Slope of Enlightenment
- Trough of Disillusionment

Speculative Locking

Correctly synchronizing a program with locks is hard

- Fine-grain locking
  - difficult to program
  - high overhead

- Coarse-grain locking
  - poor performance
  - poor scalability

- But, concurrent critical sections usually access disjoint data
  - So, they could actually run in parallel...
  - ...except that they conflict on accessing the lock variable
Speculative Lock Elision
[Rajwar & Goodman, MICRO 2001]

- Speculatively execute critical sections in parallel

- **Key Idea:** Detect & elide the lock access
  - Upon a lock acquire, don’t actually acquire lock
  - Checkpoint processor state
  - Run critical sections in parallel
  - Detect conflicting *data* accesses via coherence protocol
  - Any invalidates before lock release cause rollback
    - Then retry by acquiring lock normally

- **Advantages**
  - No locking overhead, since don’t actually acquire lock
  - Allows concurrent execution of non-conflicting critical sections.

- **How to find critical sections?**
  - Detect silent stores, ...?
Transactional Memory: The Big Idea

• Big idea I: no locks, just shared data

• Big idea II: optimistic (speculative) concurrency
  □ Execute critical section speculatively, abort on conflicts
  □ “Better to beg for forgiveness than to ask for permission”

```c
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt;
} end_transaction();```
Transactional Memory: Read/Write Sets

• **Read set**: set of shared addresses critical section reads
  - Example: `accts[37].bal, accts[241].bal`

• **Write set**: set of shared addresses critical section writes
  - Example: `accts[37].bal, accts[241].bal`

```c
struct acct_t { int bal; };  
shared struct acct_t accts[MAX_ACCT];  
int id_from, id_to, amt;  

begin_transaction();  
if (accts[id_from].bal >= amt) {  
   accts[id_from].bal -= amt;  
   accts[id_to].bal += amt;  
}  
end_transaction();
```
Transactional Memory: Begin

- **begin_transaction**
  - Take a local register checkpoint
  - Begin locally tracking read set (remember addresses you read)
    - See if anyone else is trying to write it
  - Locally buffer all of your writes (invisible to other processors)
  - Local actions only: no lock acquire

```c
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt;
} end_transaction();
```
• **end_transaction**
  - Check read set: is all data you read still valid (no writes to any)
  - Yes? Commit transactions: commit writes
  - No? Abort transaction: restore checkpoint

```c
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt;
} end_transaction();
```
Transactional Execution

**Thread 0**

\[
\begin{align*}
\text{id}_{\text{from}} &= 241; \\
\text{id}_{\text{to}} &= 37; \\
\text{begin\_transaction}(); \\
\text{if(accts}[241].\text{bal} > 100) \{ \\
\text{...} \\
\text{// write accts}[241].\text{bal} \\
\text{// abort}
\}
\end{align*}
\]

**Thread 1**

\[
\begin{align*}
\text{id}_{\text{from}} &= 37; \\
\text{id}_{\text{to}} &= 241; \\
\text{begin\_transaction}(); \\
\text{if(accts}[37].\text{bal} > 100) \{ \\
\text{accts}[37].\text{bal} &= \text{amt}; \\
\text{accts}[241].\text{bal} &= \text{amt}; \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{end\_transaction}(); \\
\text{// no writes to accts}[241].\text{bal} \\
\text{// commit}
\end{align*}
\]
Transactional Execution II (More Likely)

Thread 0

id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
    accts[241].bal -= amt;
    acts[37].bal += amt;
}
end_transaction();
// no write to accts[240].bal
// commit

Thread 1

id_from = 450;
id_to = 118;

begin_transaction();
if(accts[450].bal > 100) {
    accts[450].bal -= amt;
    acts[118].bal += amt;
}
end_transaction();
// no write to accts[450].bal
// no write to accts[118].bal
// commit

• Critical sections execute in parallel
Implementation Design Space

• Four main components:
  - Logging/buffering
    - Registers & memory
  - Conflict detection
    - Two accesses to a location, at least one is a write
  - Abort/rollback
  - Commit

Many implementation approaches
(hardware, software, hybrids)
Preserving Register Values

- Begin transaction
  - Take register checkpoint

- Commit transaction
  - Free register checkpoint

- Abort transaction
  - Restore register checkpoint
Version Management for Memory - Lazy

- Store
  - Put all writes into “write table”

- Load
  - If address in “write table”, read value from “write table”
  - Otherwise, read from memory

- Commit transaction (slow)
  - Write all entries from “write table” to memory, clear it

- Abort transaction (fast)
  - Clear “write table”
Version Management for Memory - Eager

• Store
  ❑ If address not in “write set”, then:
    ❑ 1. read old value and put it into “write log”
    ❑ 2. add address to “write set”
  ❑ Write stores directly to memory

• Load
  ❑ Read from directly from memory (fast)

• Commit transaction
  ❑ Nothing (fast)

• Abort transaction (slow)
  ❑ Traverse log, write logged values back into memory
Conflict Detection - Lazy

• Store
  ☐ Add address to “write set” (if not already present)

• Load
  ☐ Add address to “read set” (if not already present)

• Commit transaction
  ☐ For each address $A$ in “write set”
    ☐ For each other thread $T$
      ☐ If $A$ is in $T$’s “read set”, abort $T$’s transaction
Conflict Detection - Eager

• Store
  - Add address $A$ to “write set” (if not already present)
  - For each other thread $T$
    - If $A$ is in $T$’s “write set” or “read set”, trigger conflict

• Load
  - Add address to “read set” (if not already present)
  - For each other thread $T$
    - If $A$ is in $T$’s write set, trigger conflict

• Conflict: abort either transaction

• Commit transaction
  - Ok if not yet aborted, just clear read and write sets
Software Transactional Memory (STM)

• Add extra software to perform TM operations
• Version management
  □ Software data structure for log or write table
  □ Eager or lazy
• Conflict detection
  □ Software data structure (lock table), mostly lazy
  □ “object” or “block” granularity
• Commit
  □ Need to ensure atomic update of all state
  □ Grabs lots of locks, or a global commit lock
• Many possible implementations & semantics
Hardware Transactional Memory (HTM)

- Leverage invalidation-based cache coherence
  - Each cache block has “read-only” or “read-write” state
  - Coherence invariant:
    - Many “read-only” (shared) blocks — or —
    - Single “read-write” block

- Add pair of bits per cache block: “read” & “write”
  - Set on loads/stores during transactional execution
  - If another core steals block from cache, abort
    - Read or write request to block with “write” bit set
    - Write request to block with “read” bit set

- Low-overhead conflict detection...
  - But only if all blocks fit in cache
HTM vs STM

- Hardware transactional memory (HTM)
  - Requires hardware (Intel Haswell has Tx support)
  - Simple for “bounded” case
  - Unbounded TM in hardware really complicated
    - Size: tracking conflicts after cache overflow
    - Duration: context switching transactions
  - Cache block granularity for conflicts

- Software transactional memory (STM)
  - Here today (prototype compilers from Intel & others)
  - Generally “weaker” semantics
  - Slow (2x or more single-thread overhead)
    - Lots of extra instructions on memory operations
Hybrid Transactional Memory

• Hardware-accelerated STM
  ▫ Add special hardware tracking features
  ▫ Under control of software
  ▫ Can reduce STM overhead, but perhaps not enough

• Hybrid HTM/STM
  ▫ Use HTM mode most of the time
  ▫ Resort to STM only on overflows and such
  ▫ Getting the interaction right is actually really tricky
So, Let’s Just Do Transactions?

• What if...
  - Read-set or write-set bigger than cache?
  - Transaction gets swapped out in the middle?
  - Transaction wants to do I/O or SYSCALL (not-abortable)?

• How do we transactify existing lock based programs?
  - Replace acquire with begin_trans does not always work

• Several different kinds of transaction semantics
  - Are transactions atomic relative to code outside of transactions?
Transactions ≠ Critical Sections

What is wrong with this program?

\[
\begin{align*}
\text{begin\_transaction();} \\
\text{flagA = true;} \\
\text{while (!flagB) {}} \\
\text{//update m} \\
\text{end\_transaction();}
\end{align*}
\]

\[
\begin{align*}
\text{begin\_transaction();} \\
\text{while (!flagA) {}} \\
\text{flagB = true;} \\
\text{//update n} \\
\text{end\_transaction();}
\end{align*}
\]

A less contrived example...

```
begin_transaction();
... 
queueA->enqueue(val1);
while (queueB->empty()){}
//access queueB 
... 
end_transaction();
```

```
begin_transaction();
... 
queueB->enqueue(val2);
while (queueA->empty()){}
//access queueA 
... 
end_transaction();
```