EECS 570
Lecture 7
Transactional Memory
Winter 2022
Prof. Yatin Manerkar

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.
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

Step 1

Step 2

Step 3
Global Synchronization Barrier

- At a barrier
  - All threads wait until all other threads have reached it
- Strawman implementation (**wrong**!)
  
  ```
  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:

```
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...

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

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

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, buffering speculative state
  - Detect conflicting data accesses via coherence protocol
  - Any invalidates before lock release cause rollback, otherwise commit
    - 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 particular instruction sequences
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**

```
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: End

- **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

```
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 I (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
// no write to accts[37].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
.Transactional Execution II (Conflict)

Thread 0

```
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
  ...
  // write accts[241].bal
  // abort
}
```

Thread 1

```
id_from = 37;
id_to = 241;

begin_transaction();
if(accts[37].bal > 100) {
  accts[37].bal -= amt;
  acts[241].bal += amt;
}
end_transaction();
// no writes to accts[241].bal
// no writes to accts[37].bal
// commit
```
Implementation Design Space

- Four main components:
  - Logging/buffering/Version Management
    - 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 \(\text{(fast)}\)

- **Commit transaction**
  - Nothing \(\text{(fast)}\)

- **Abort transaction** \(\text{(slow)}\)
  - 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)
  □ 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
TM for Performance

Intel Haswell

Applying Intel® TSX

Application with Coarse Grain Lock

Application re-written with Finer Grain Locks

An example of secondary benefits of Intel® TSX

Graph showing scaling vs. threads for Coarse Grain Lock + Intel® TSX and Fine Grain Locks + Intel® TSX.
But, more important benefit of TM is programmability

Performance of fine-grained locks
Simplicity of using one coarse-grained lock

Unlike locks, transactions are typically composable
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)?

• HTM is not easy to do correctly
  - 2014 bug in Intel TSX for Haswell and Broadwell
    - Fixed by turning off TM for the affected processors
    - 2021: disabling TM on more processors due to other bugs found

• 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?
  - Interactions with non-transactional code can cause issues
    - e.g. [Chong et al. PLDI 2018]
Transactions ≠ Critical Sections

What is wrong with this program?

begin_transaction();
flagA = true;
while (!flagB) {}
//update m
end_transaction();

begin_transaction();
while (!flagA) {}
flagB = true;
//update n
end_transaction();

A less contrived example...

Queue* queueA = new Queue();
Queue* queueB = new Queue();

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

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