EECS 570
Lecture 9
Snooping Coherence
Winter 2016
Prof. Thomas Wenisch

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

Slides developed in part by Profs. Falsafi, Hardavellas, Nowatzyk, and Wenisch of EPFL, Northwestern, CMU, U-M.
Readings

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

For Wednesday 2/16:
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?
  
  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
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();
```
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

```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

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;
  accts[241].bal += amt;
}
end_transaction();
// no writes to accts[241].bal
// no writes to accts[37].bal
// commit
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
// 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
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_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();
```
Unit 3 - Cache Coherence & Memory Consistency
Cache Coherence

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)
No-Cache, No-Problem

Processor 0
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash

Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash

• Scenario I: processors have no caches
  □ No problem
Cache Incoherence

- Scenario II: processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
  - Can get incoherent (inconsistent)
Snooping Cache-Coherence Protocols

Bus provides serialization point

Each cache controller “snoops” all bus transactions

- take action to ensure coherence
  - invalidate
  - update
  - supply value
- depends on state of the block and the protocol
Scalable Cache Coherence

• Scalable cache coherence: two part solution

• Part I: bus bandwidth
  - Replace non-scalable bandwidth substrate (bus)...
  - ...with scalable bandwidth one (point-to-point network, e.g., mesh)

• Part II: processor snooping bandwidth
  - Interesting: most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)...
  - ...with scalable directory protocol (only spam processors that care)

• We will cover this in Unit 3
Approaches to Cache Coherence

• Software-based solutions
  ❑ Mechanisms:
    ❖ Mark cache blocks/memory pages as cacheable/non-cacheable
    ❖ Add “Flush” and “Invalidate” instructions
    ❖ When are each of these needed?
  ❑ Could be done by compiler or run-time system
  ❑ Difficult to get perfect (e.g., what about memory aliasing?)
  ❑ Will revisit this briefly in Unit 3...

• Hardware solutions are far more common
  ❑ In Unit 2, we study schemes that rely on broadcast over a bus
**Write-Through Scheme 1: Valid-Invalid Coherence**

*Valid-Invalid Coherence*

- Allows multiple readers, but must write through to bus
  - Write-through, no-write-allocate cache
- All caches must monitor (aka “snoop”) all bus traffic
  - Simple state machine for each cache frame

---

**t1:** Store A=1

**t2:** BusWr A=1

**t3:** Invalidate A
Valid-Invalid Snooping Protocol

Actions:
- Ld, St, BusRd, BusWr
- Write-through, no-write-allocate cache
- 1 bit of storage overhead per cache frame
Write Through Scheme 2: Write-Update Coherence

- **t1:** Store $A=1$
- **t2:** BusWr $A=1$
- **t3:** Snarf $A$

**Write-Update Coherence**

- Instead of invalidation, “Snarf” new value of $A$ off the Bus
- But, 15% of cache accesses are stores
  - Tremendous bus and cache tag BW requirement
Supporting Write-Back Caches

- Write-back caches drastically reduce bus write bandwidth

- Key idea: add notion of “ownership” to Valid-Invalid
  - Mutual exclusion – when “owner” has only replica of a cache block, it may update it freely
  - Sharing – multiple readers are ok, but they may not write without gaining ownership

- Need to find which cache (if any) is an owner on read misses
- Need to eventually update memory so writes are not lost
Modified-Shared-Invalid (MSI) Protocol

• Three states tracked per-block at each cache
  ❑ Invalid – cache does not have a copy
  ❑ Shared – cache has a read-only copy; clean
    ○ Clean == memory is up to date
  ❑ Modified – cache has the only copy; writable; dirty
    ○ Dirty == memory is out of date

• Three processor actions
  ❑ Load, Store, Evict

• Five bus messages
  ❑ BusRd, BusRdX, BusInv, BusWB, BusReply
  ❑ Could combine some of these
Modified-Shared -Invalid (MSI) Protocol

Load / BusRd

Invalid → Shared

1: Load A

P1

A [↓ S]: 0

2: BusRd A

Bus

P2

A [I]

3: BusReply A

A: 0
Modified-Shared -Invalid (MSI) Protocol

Invalid

Load / BusRd

Shared

BusRd / [BusReply]

Load / --

1: Load A

P1

A [S]: 0

3: BusReply A

Bus

A: 0

P2

A [↓ S]: 0

2: BusRd A
Modified-Shared -Invalid (MSI) Protocol

Invalid

Load / BusRd

Shared

Evict /

BusRd / [BusReply]

Load / --

Evict A
Modified-Shared -Invalid (MSI) Protocol

- **Invalid**
  - Load / BusRd
  - BusRdX / [BusReply]
  - Evict / --

- **Shared**
  - BusRd / [BusReply]
  - Load / --

- **Modified**
  - Store / BusRdX

- **Load, Store / --**

1: Store A
2: BusRdX A
3: BusReply A

**P1**
- A [S I]: 0

**P2**
- A [† M]: 0 1

**Bus**
- A: 0
Modified-Shared -Invalid (MSI) Protocol

Invalid
- Load / BusRd
- BusRdX / [BusReply]
- Evict / --
- Store / BusRdX
- Load, Store / --

Shared
- Load / BusRd
- BusRd / BusReply
- Evict / --

Modified
- BusRd / BusReply

Bus
- A [S]: 1
- A [M]: 1

P1
- 1: Load A
- 2: BusRd A
- 4: Snarf A

P2
- 1: Load A
- 2: BusRd A
- 3: BusReply A

A: Ø 1
Modified-Shared -Invalid (MSI) Protocol

- Invalid
  - BusRdX, BusInv / [BusReply]
  - Evict / --
  - Store / BusRdX

- Modified
  - Load, Store / --

- Shared
  - Load / BusRd
  - BusRd / BusReply
  - BusRdX, BusInv / [BusReply]
  - Evict / --
  - Store / BusInv

- Bus
  - A [S M]: 2
  - A [S I]
  - A: 1

1: Store A aka "Upgrade"
2: BusInv A

P1

EECS 570
Modified-Shared -Invalid (MSI) Protocol

Invalid

Load / BusRd

BusRdX, BusInv / [BusReply]

Shared

Load / BusRd

Evict / --

Store / BusRdX

BusRdX / BusReply

Modified

BusRd / BusReply

Store / BusInv

Load, Store / --

A:

P1

A [M I]: 2

P2

A [I M]: 3

Bus

3: BusReply A

1: Store A

2: BusRdX A
Modified-Shared -Invalid (MSI) Protocol

- Invalid
  - Store / BusRdX
  - Evict / BusInv / [BusReply]
  - Load / BusRd

- Modified
  - BusRdX, BusInv / [BusReply]
  - Evict / --
  - Store / BusInv

- Shared
  - BusRd / BusReply
  - Evict / --
  - Load / BusRd / [BusReply]

- Load, Store / --

1: Evict A
2: BusWB A

A [l]
A [M l]: 3

Bus
A: ± 3

P1
P2
**MSI Protocol Summary**

```
Invalied
  Store / BusRdX
  BusRdX, BusInv / [BusReply]
  Load / BusRd

Shared
  BusRd / BusReply
  Load / BusRd
  BusRdX / [BusReply]

Modified
  Evict / BusWB
  Store / BusInv
  BusRd / BusReply

Cache Actions:
- Load, Store, Evict

Bus Actions:
- BusRd, BusRdX
  BusInv, BusWB, BusReply
```
Update vs. Invalidate

• Invalidation is bad when:
  □ Single producer and many consumers of data

• Update is bad when:
  □ Multiple writes by one CPU before read by another
  □ Junk data accumulates in large caches (e.g., process migration)
Coherence Decoupling
[Huh, Chang, Burger, Sohi ASPLOS04]

• After invalidate, keep stale data around
  □ On subsequent read, speculatively supply stale value
  □ Confirm speculation with a normal read operations
  □ Need a branch-prediction-like rewind mechanism
  □ Completely solves false sharing problem
  □ Also addresses “silent”, “temporally-silent” stores

• Can use update-like mechanisms to improve prediction
  □ Paper explores a variety of update heuristics
  □ E.g., piggy-back value of 1st write on invalidation message
MESI Protocol (aka Illinois)

- MSI suffers from frequent read-upgrade sequences
  - Leads to two bus transactions, even for private blocks
  - Uniprocessors don’t have this problem

- Solution: add an “Exclusive” state
  - Exclusive – only one copy; writable; clean
    - Can detect exclusivity when memory provides reply to a read
  - Stores transition to Modified to indicate data is dirty
    - No need for a BusWB from Exclusive
Mesi Protocol Summary

- **Invalid**
  - Load / BusRd (reply from mem)
  - BusRdX, BusInv / [BusReply]
  - Evict / --
  - Load / BusRd (reply from mem)

- **Shared**
  - Load / --
  - BusRdX, BusInv / [BusReply]
  - Evict / --
  - Store / --

- **Exclusive**
  - BusRdX / BusReply
  - Store / BusInv
  - BusRd / BusReply
  - Evict / BusWB

- **Modified**
  - Load, Store / --
  - BusRd / [BusReply]
  - Store / BusInv
  - BusRdX / BusReply
  - Evict / BusWB
MOESI Protocol

- MESI must write-back to memory on $M \rightarrow S$ transitions
  - Because protocol allows “silent” evicts from shared state, a dirty block might otherwise be lost
  - But, the writebacks might be a waste of bandwidth
    - E.g., if there is a subsequent store
    - Common case in producer-consumer scenarios

- Solution: add an “Owned” state
  - Owned – shared, but dirty; only one owner (others enter $S$)
    - Entered on $M \rightarrow S$ transition, aka “downgrade”
  - Owner is responsible for writeback upon eviction
MOESI Framework

[Sweazey & Smith ISCA86]

M - Modified (dirty)
O - Owned (dirty but shared)  WHY?
E - Exclusive (clean unshared) only copy, not dirty
S - Shared
I - Invalid

Variants
- MSI
- MESI
- MOSI
- MOESI

ownership
validity
exclusiveness
DEC Firefly

• An update protocol for write-back caches

• States
  □ Exclusive – only one copy; writeable; clean
  □ Shared – multiple copies; write hits write-through to all sharers and memory
  □ Dirty – only one copy; writeable; dirty

• Exclusive/dirty provide write-back semantics for private data

• Shared state provides update semantics for shared data
  □ Uses “shared line” bus wire to detect sharing status

• Well suited to producer-consumer; process migration hurts
DEC Firefly Protocol Summary

- **Exclusive**
  - Load Miss & !SL
  - Store & !SL / --
- **Shared**
  - Load Miss & SL
  - BusRd, BusWr / BusReply
  - Store & SL / BusWr
- **Dirty**
  - Load, Store / --
  - BusRd / BusReply (update mem)
  - BusWr / snarf
Non-Atomic State Transitions

Operations involve multiple actions
- Look up cache tags
- Bus arbitration
- Check for writeback
- Even if bus is atomic, overall set of actions is not
- Race conditions among multiple operations

Suppose P1 and P2 attempt to write cached block A
- Each decides to issue BusUpgr to allow S → M

Issues
- Handle requests for other blocks while waiting to acquire bus
- Must handle requests for this block A

We will revisit this at length in Unit 3
Scalability problems of Snoopy Coherence

- **Prohibitive bus bandwidth**
  - Required bandwidth grows with # CPUS...
  - ... but available BW per bus is fixed
  - Adding busses makes serialization/ordering hard

- **Prohibitive processor snooping bandwidth**
  - All caches do tag lookup when ANY processor accesses memory
  - Inclusion limits this to L2, but still lots of lookups

- **Upshot**: bus-based coherence doesn’t scale beyond 8–16 CPUs