EECS 570
Lecture 9
Snooping Coherence
Winter 2018
Prof. Satish Narayanasamy

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 2/7:

- Sorin et al - A Primer on Memory Consistency and Cache Coherence, Ch. 8
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;
    acts[241].bal += amt;
}
end_transaction();
// no writes to accts[241].bal
// no writes to accts[37].bal
// commit
Transactional Execution II (More Likely)

- 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
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`)

**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
No-Cache, No-Problem

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)
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

- **t1**: Store A=1
- **t2**: BusWr A=1
- **t3**: Invalidate A

**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
Valid-Invalid Snooping Protocol

Actions:
Ld, St, BusRd, BusWr

Write-through, no-write-allocate cache

1 bit of storage overhead per cache frame

Valid

Invalid

Load / --

Store / BusWr

Load / BusRd

BusWr

Store / BusWr
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

3: BusReply A

A: 0

P2

A [↓]

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

Invalid \(\xrightarrow{\text{Load / BusRd}}\) Shared

Shared \(\xrightarrow{\text{BusRd} / [\text{BusReply}]}\) 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

BusRd / [BusReply]

Evict / --

Load / --

Evict A

P1

A [S]: 0

P2

A [S I]

Bus

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

Invalid → Load / BusRd → BusRdX / [BusReply] → Modified

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

Store / BusRdX

Modified → Evict / --

Load, Store / --

P1

A [S I]: 0

3: BusReply A

P2

A [T M]: 0 1

2: BusRdX A

Bus

A: 0

EECS 570

Lecture 9 Slide 34
**Modified-Shared -Invalid (MSI) Protocol**

Invalid --- Load / BusRd

Invalid --- BusRdX / [BusReply]

Modified --- Store / BusRdX

Modified --- BusRd / BusReply

Shared --- Evict / --

Shared --- BusRd / [BusReply]

Load / --

Load, Store / --

---

**Process Diagram**

1: Load A

P1

2: BusRd A

Bus

3: BusReply A

P2

4: Snarf A

A [↓ S]: 1

A [M S]: 1

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

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

- Shared
  - BusRd / BusReply
  - Store / BusInv
  - BusReply

- Modified
  - Store / BusRdX

- Load, Store / --

1: Store A aka “Upgrade”
- P1
  - A [S M]: 2

2: BusInv A
- P2
  - A [S I]

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

Invalid

Modified

Load / BusRd

Load, Store / --

BusRdX, BusInv / [BusReply]

BusRdX / BusReply

Store / BusInv

Evict / --

BusRd / BusReply

BusRd / [BusReply]
Modified-Shared -Invalid (MSI) Protocol

Invalid

Modified

Shared

Load / BusRd

BusRdX, BusInv / [BusReply]

Evict / --

Store / BusInv

BusRd / BusReply

BusRdX / BusReply

BusRdX, BusInv / [BusReply]

Evict / --

Load / --

Store / BusRdX

BusRd / [BusReply]

P1

A [I]

P2

A [M I]: 3

Bus

A: 1 3

1: Evict A

2: BusWB A
MSI Protocol Summary

Cache Actions:
- Load, Store, Evict

Bus Actions:
- BusRd, BusRdX
- BusInv, BusWB, BusReply

Invalid

Shared

Modified

Load / BusRd

BusRdX, BusInv / [BusReply]

Evict / --

Evict / --

Store / BusRdX

BusRdX / BusReply

Store / BusInv

Load, Store / --

Load / --

BusRd / [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 1\textsuperscript{st} 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 cache)
  - BusRdX, BusInv / [BusReply]
  - Evict / --
  - Load / BusRd (reply from mem)

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

- **Exclusive**
  - BusRd / BusReply
  - BusRdX / BusRdX
  - Evict / BusWB
  - Load / --

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

- MESI must write-back to memory on $M \to S$ 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 $M \to S$ 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

Lecture 9 Slide 45
DEC Firefly

- An update protocol for write-back caches
- States
  - Exclusive – only one copy; writable; clean
  - Shared – multiple copies; write hits write-through to all sharers and memory
  - Dirty – only one copy; writable; 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**:
  - Store & !SL / --
  - BusRd, BusWr / BusReply

- **Shared**:  
  - Load Miss & SL
  - BusWr / snarf
  - BusRd / BusReply (update mem)
  - BusWr / snarf

- **Dirty**:  
  - Load, Store / --
  - Load Miss & !SL
  - BusRd / BusReply

**States**: Exclusive, Shared, Dirty

**Actions**:
- Store
- Load Miss & SL
- Load, Store / --
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