EECS 570
Lecture 9
Snooping Coherence
Winter 2017
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:
- Daniel J. Sorin, Mark D. Hill, and David A. Wood, 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

\[ \text{id}_\text{from} = 241; \]
\[ \text{id}_\text{to} = 37; \]

\[ \text{begin}\_\text{transaction}(); \]
\[ \text{if} (\text{accts}[241].\text{bal} > 100) \{ \]
\[ \quad \ldots \]
\[ \quad \text{// write accts[241].bal} \]
\[ \quad \text{// abort} \]

Thread 1

\[ \text{id}_\text{from} = 37; \]
\[ \text{id}_\text{to} = 241; \]

\[ \text{begin}\_\text{transaction}(); \]
\[ \text{if} (\text{accts}[37].\text{bal} > 100) \{ \]
\[ \quad \text{accts}[37].\text{bal} -= \text{amt}; \]
\[ \quad \text{acts}[241].\text{bal} += \text{amt}; \]
\[ \}
\[ \text{end}\_\text{transaction}(); \]
\[ \text{// no writes to accts[241].bal} \]
\[ \text{// no writes to accts[37].bal} \]
\[ \text{// commit} \]
**Transactional Execution II (More Likely)**

- Critical sections execute in parallel

```c
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
```
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`)
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)
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

- 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
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

P2

A [↓]

3: BusReply A

Bus

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

Invalid \rightarrow \text{Load} / \text{BusRd} \rightarrow \text{Shared}

\text{BusRd} / [\text{BusReply}] \rightarrow \text{Load} / --

1: \text{Load A} \rightarrow \text{P1} \rightarrow A [S]: 0

2: \text{BusRd A} \rightarrow \text{A} [\uparrow S]: 0

3: \text{BusReply A} \rightarrow \text{Bus} \rightarrow A: 0

1: \text{Load A} \rightarrow \text{P2}
Modified-Shared -Invalid (MSI) Protocol

Invalid <--> Load / BusRd <--> Shared

Shared <--> Evict / --

BusRd / [BusReply] <--> Load / --

P1
A [S]: 0

P2
A [S I]

Bus
A: 0

Evict A
Modified-Shared -Invalid (MSI) Protocol

Invalid → Load / BusRd

BusRdX / [BusReply] → Shared

Load / -- → BusRd / [BusReply]

Evict / --

Modified

Store / BusRdX

Load, Store / --
Modified-Shared -Invalid (MSI) Protocol

- **Invalid**: Load / BusRd
- **Shared**: BusRdX / [BusReply]
- **Modified**: Store / BusRdX
- **Evict**: BusRd / BusReply

**Process Flow**

1. **Load A**
   - **P1**: A \([\uparrow S]: 1\)
   - **Bus**: A: 0
   - **P2**: A \([M \uparrow S]: 1\)
   - **BusReply A**: A: 0
2. **BusRd A**
3. **BusReply A**
4. **Snarf A**
Modified-Shared -Invalid (MSI) Protocol

- **Invalid**
  - Load / BusRd
  - BusRdX, BusInv / [BusReply] 
  - Store / BusRdX

- **Shared**
  - Load / --
  - BusRd / [BusReply]
  - Evict / --
  - BusRd / BusReply
  - Store / BusInv

- **Modified**
  - Load, Store / --

- **Bus**
  - A [S M]: 2
  - A [S I]
  - 1: Store A aka “Upgrade”
  - 2: BusInv A

- **P1**
  - A: 1

- **P2**
  - A [S I]
Modified-Shared -Invalid (MSI) Protocol

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

P1:
- A [M I]: 2

P2:
- A [I M]: 3
  - 2: BusRdX A

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

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

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

- **Modified**
  - Load, Store / --
  - BusRdX / BusReply
  - Evict / BusWB

**Transactions:**

- **P1**
  - A [I]
  - A [M I]: 3
  - BusWB A

- **P2**
  - A: ± 3
MSI Protocol Summary

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 / --

Shared

- Load / BusRd (reply from cache)
- Evict / --
- Store / BusInv
- BusRd / BusReply

Exclusive

- BusRdX / BusReply
- Evict / BusWB
- Store / --
- Load / --

Modified

- Store / BusInv
- BusRdX / BusReply
- Load, Store / --
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
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
  - BusRd, BusWr / BusReply
  - Store & !SL / --

- **Shared**
  - Load Miss & SL
  - BusRd / BusReply
  - BusWr / snarf
  - Store & SL / BusWr

- **Dirty**
  - Store
  - BusWr / snarf
  - BusRd / BusReply (update mem)

- **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 \rightarrow 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