Admin

- Work on Projects
- HW #5 Assigned
- May skip Virtual Machines/Security unit (papers)
This Unit: Shared Memory Multiprocessors

- Three issues
  - Cache coherence
  - Synchronization
  - Memory consistency

- Two cache coherence approaches
  - “Snooping” (SMPs): < 16 processors
  - “Directory”/Scalable: lots of processors

Directory Coherence Protocols

- Observe: physical address space statically partitioned
  - Can easily determine which memory module holds a given line
    - That memory module sometimes called “home”
  - Can’t easily determine which processors have line in their caches
    - Bus-based protocol: broadcast events to all processors/caches
      - Simple and fast, but non-scalable

- **Directories**: non-broadcast coherence protocol
  - Extend memory to track caching information
  - For each physical cache line whose home this is, track:
    - **Owner**: which processor has a dirty copy (i.e., M state)
    - **Sharers**: which processors have clean copies (i.e., S state)
  - Processor sends coherence event to home directory
    - Home directory only sends events to processors that care
MSI Directory Protocol

- **Processor side**
  - Directory follows its own protocol (obvious in principle)

- **Similar to bus-based MSI**
  - Same three states
  - Same five actions (keep BR/BW names)
  - Minus grayed out arcs/actions
    - Bus events that would not trigger action anyway
    - Directory won’t bother you unless you need to act

![Diagram](image)

Directory MSI Protocol

- **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`

- **P0**
  - `S:500`
  - `M:400`
  - `S:0:500`
  - `S:400`
  - `M:300`
  - `M:1:400`

- **P1**
  - `S:0:500`
  - `S:0:1:400`

- **Directory**
  - `S:500`
  - `M:400`
  - `S:0:500`

- **1d** by P1 sends BR to directory
  - Directory sends BR to P0, P0 sends P1 data, does WB, goes to S

- **st** by P1 sends BW to directory
  - Directory sends BW to P0, P0 goes to I
Directory Flip Side: Latency

• Directory protocols
  + Lower bandwidth consumption → more scalable
  − Longer latencies

• Two read miss situations
  ➢ Unshared block: get data from memory
    ▪ Bus: 2 hops (P0 → memory → P0)
    ▪ Directory: 2 hops (P0 → memory → P0)
  ➢ Shared or exclusive block: get data from other processor (P1)
    ▪ Assume cache-to-cache transfer optimization
    ▪ Bus: 2 hops (P0 → P1 → P0)
    − Directory: 3 hops (P0 → memory → P1 → P0)
    ▪ Common, with many processors high probability someone has it

Directory Flip Side: Complexity

• Latency not only issue for directories
  ➢ Subtle correctness issues as well
  ➢ Stem from unordered nature of underlying inter-connect

• Individual requests to single cache line must appear atomic
  ➢ Bus: all processors see all requests in same order
    ▪ Atomicity automatic
  ➢ Point-to-point network: requests may arrive in different orders
    ▪ Directory has to enforce atomicity explicitly
    ▪ Cannot initiate actions on request B...
    ▪ Until all relevant processors have completed actions on request A
    ▪ Requires directory to collect acks, queue requests, etc.

• Directory protocols
  ➢ Obvious in principle
    ▪ Extremely complicated in practice
One Down, Two To Go

- Coherence only one part of the equation
  - Synchronization
  - Consistency

The Need for Synchronization

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1,accts,r3</td>
<td>0: addi r1,accts,r3</td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td>1: ld 0(r3),r4</td>
</tr>
<tr>
<td>2: blt r4,r2,6</td>
<td>2: blt r4,r2,6</td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td>3: sub r4,r2,r4</td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>4: st r4,0(r3)</td>
</tr>
<tr>
<td>5: call spew_cash</td>
<td>5: call spew_cash</td>
</tr>
</tbody>
</table>

- We’re not done, consider the following execution
  - Write-back caches (doesn’t matter, though), MSI protocol
- What happened?
  - We got it wrong ... and coherence had nothing to do with it
The Need for Synchronization

• What really happened?
  ➢ Access to `accts[241].bal` should conceptually be **atomic**
    ▪ Transactions should not be "interleaved"
    ▪ But that’s exactly what happened
    ▪ Same thing can happen on a multiprogrammed uniprocessor!

• Solution: **synchronize** access to `accts[241].bal`

```
Processor 0                     Processor 1
0: addi r1,accts,r3            0: addi r1,accts,r3
1: ld 0(r3),r4                 1: ld 0(r3),r4
2: blt r4,r2,6                 2: blt r4,r2,6
3: sub r4,r2,r4                3: sub r4,r2,r4
4: st r4,0(r3)                 4: st r4,0(r3)
5: call spew_cash              5: call spew_cash
                              S:500 500
                              S:500 S:500 500
                              M:400 400
                              I: 400 400
```

Synchronization

• **Synchronization**: second issue for shared memory
  ➢ Regulate access to shared data
  ➢ Software constructs: semaphore, monitor
  ➢ Hardware primitive: **lock**
    ▪ Operations: `acquire(lock)` and `release(lock)`
    ▪ Region between `acquire` and `release` is a **critical section**
    ▪ Must interleave `acquire` and `release`
    ▪ Second consecutive `acquire` will fail (actually it will block)

```
struct acct_t { int bal; };  // critical section
shared struct acct_t accts[MAX_ACCT];
shared int lock;
int id,amt;
acquire(lock);
if (accts[id].bal >= amt) {
  accts[id].bal -= amt;
  spew_cash();
}
release(lock);
```
Working Spinlock: Test-And-Set

- ISA provides an atomic lock acquisition instruction
  - Example: **test-and-set**
    
    \[
    \text{t\&s} \ r1,0(\&\text{lock})
    \]

  - Atomically executes
    \[
    \text{ld} \ r1,0(\&\text{lock})
    \text{st} \ 1,0(\&\text{lock})
    \]
  - If lock was initially free (0), acquires it (sets it to 1)
  - If lock was initially busy (1), doesn’t change it

- New acquire sequence
  
  \[
  \begin{align*}
  \text{A0:} & \quad \text{t\&s} \ r1,0(\&\text{lock}) \\
  \text{A1:} & \quad \text{bnez} \ r1,\#\text{A0}
  \end{align*}
  \]

- More general atomic mechanisms
  - **swap, exchange, fetch-and-add, compare-and-swap**

Test-and-Set Lock Correctness

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0: t&amp;s r1,0(&amp;lock)</td>
<td>A0: t&amp;s r1,0(&amp;lock)</td>
</tr>
<tr>
<td>A1: bnez r1,#A0</td>
<td>A0: t&amp;s r1,0(&amp;lock)</td>
</tr>
<tr>
<td>CRITICAL SECTION</td>
<td>A1: bnez r1,#A0</td>
</tr>
<tr>
<td></td>
<td>A0: t&amp;s r1,0(&amp;lock)</td>
</tr>
<tr>
<td></td>
<td>A1: bnez r1,#A0</td>
</tr>
</tbody>
</table>

+ Test-and-set lock actually works
  - Processor 1 keeps spinning
**RISC Test-And-Set**

- **t&s**: a load and store in one insn is not very “RISC”
  - Broken up into micro-ops, but then how is it made atomic?

- **ll/sc**: load-locked / store-conditional
  - Atomic load/store pair
    - `ll r1,0(&lock)`
    - // potentially other insns
    - `sc r2,0(&lock)`
  - On `ll`, processor remembers address...
    - ...And looks for writes by other processors
    - If write is detected, next `sc` to same address is annulled
      - Sets failure condition

---

**“Test-and-Set” Lock Performance**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0: t&amp;s  r1,0(&amp;lock)</td>
<td>A0: t&amp;s  r1,0(&amp;lock)</td>
</tr>
<tr>
<td>A1: bnez r1,#A0</td>
<td>A0: t&amp;s  r1,0(&amp;lock)</td>
</tr>
<tr>
<td>A0: t&amp;s r1,0(&amp;lock)</td>
<td>A1: bnez r1,#A0</td>
</tr>
<tr>
<td>A1: bnez r1,#A0</td>
<td>A0: t&amp;s r1,0(&amp;lock)</td>
</tr>
<tr>
<td></td>
<td>A1: bnez r1,#A0</td>
</tr>
</tbody>
</table>

- ...but performs poorly
  - Consider 3 processors rather than 2
  - Processor 2 (not shown) has the lock and is in the critical section
  - But what are processors 0 and 1 doing in the meantime?
    - Loops of `t&s`, each of which includes a `st`
    - Repeated stores by multiple processors costly
    - Generating a ton of useless interconnect traffic
Test-and-Test-and-Set Locks

- Solution: **test-and-test-and-set locks**
  - New acquire sequence
    - A0: `ld r1,0(&lock)`
    - A1: `bnez r1,A0`
    - A2: `addi r1,1,r1`
    - A3: `t&S r1,0(&lock)`
    - A4: `bnez r1,A0`
  - Within each loop iteration, before doing a `t&S`
    - Spin doing a simple test (`ld`) to see if lock value has changed
    - Only do a `t&S` (`st`) if lock is actually free
  - Processors can spin on a busy lock locally (in their own cache)
    - Less unnecessary interconnect traffic
  - Note: test-and-test-and-set is not a new instruction!
    - Just different software

Lock Problems

- T&T&S: Release -> many cores want lock
- Software Queues/Trees for scalable locks
  - Must be fair
  - Overhead if no contention

- Programming with Locks is..... Tricky
  - Need correct & highly concurrent
  - Lock granularity is issue:
    - Lock individual words vs. lock entire data structure
  - Multiple Locks...P1: L1 then L2; P2: L2 then L1 .... **DEADLOCK!**
Research: Transactional Memory (TM)

- **Transactional Memory**
  - Programming simplicity of coarse-grain locks
  - Higher concurrency (parallelism) of fine-grain locks
    - Critical sections only serialized if data is actually shared
  - No lock acquisition overhead
  - Hottest thing since sliced bread (or was a few years ago)
  - No fewer than nine research projects:
    - Brown, Stanford, MIT, Wisconsin, Texas, Penn, Rochester, Sun, Intel

Transactional Memory: The Big Idea

- Big idea I: **no locks, just shared data**
  - Look ma, no locks
- 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 (i.e., 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 Memory Implementation

- How are read-set/write-set implemented?
  - Track locations accessed using bits in the cache

- Read-set: additional “transactional read” bit per block
  - Set on reads between begin_transaction and end_transaction
  - Any other write to block with set bit triggers abort
  - Flash cleared on transaction abort or commit

- Write-set: additional “transactional write” bit per block
  - Set on writes between begin_transaction and end_transaction
  - Before first write, if dirty, initiate writeback (“clean” the block)
  - Flash cleared on transaction commit
  - On transaction abort: blocks with set bit are invalidated
Tricky Shared Memory Examples

- Answer the following questions:
  - **Initially: all variables zero** (that is, x is 0, y is 0, flag is 0, A is 0)
  - What value pairs can be read by the two loads? (x, y) pairs:

    ```
    thread 1 thread 2
    load x    store 1 → y
    load y    store 1 → x
    ```

  - What value pairs can be read by the two loads? (x, y) pairs:

    ```
    thread 1 thread 2
    store 1 → y    store 1 → x
    load x        load y
    ```

  - What value can be read by "Load A" below?

    ```
    thread 1 thread 2
    store 1 → A    while(flag == 0) { }    load A
    store 1 → flag
    ```

Hiding Store Miss Latency

- Recall (back from caching unit)
  - Hiding store miss latency
  - How? Store buffer

- Said it would complicate multiprocessors
  - Yes. It does.
Recall: Write Misses and Store Buffers

- Read miss?
  - Load can’t go on without the data, it must stall

- Write miss?
  - Technically, no instruction is waiting for data, why stall?

- **Store buffer**: a small buffer
  - Stores put address/value to write buffer, *keep going*
  - Store buffer writes stores to D$ in the background
  - Loads must search store buffer (in addition to D$)
  - Eliminates stalls on write misses (mostly)
  - Creates some problems (later)

- Store buffer vs. writeback-buffer
  - Store buffer: “in front” of D$, for hiding store misses
  - Writeback buffer: “behind” D$, for hiding writebacks

Memory Consistency

- **Memory coherence**
  - Creates globally uniform (consistent) view...
  - Of a single memory location (in other words: cache line)
    - Not enough
      - Cache lines A and B can be individually consistent...
      - But inconsistent with respect to each other

- **Memory consistency**
  - Creates globally uniform (consistent) view...
  - Of all memory locations relative to each other

- Who cares? Programmers
  - Globally inconsistent memory creates mystifying behavior
Coherence vs. Consistency

- **Intuition says**: P1 prints A=1
- **Coherence says?**
  - Absolutely nothing!
    - P1 can see P0’s write of flag before write of A!!! How?
      - Maybe coherence event of A is delayed somewhere in network
      - Maybe P0 has a coalescing write buffer that reorders writes
    - Imagine trying to figure out why this code sometimes “works” and sometimes doesn’t
    - **Real systems** act in this strange manner

```c
A=flag=0;
Processor 0
A=1; while (!flag); // spin
flag=1;
Processor 1
print A;
```

Store Buffers & Consistency

- Consider the following execution:
  - Processor 0’s write to A, misses the cache. Put in store buffer
  - Processor 0 keeps going
  - Processor 0 write “1” to flag hits, completes
  - Processor 1 reads flag... sees the value “1”
  - Processor 1 exits loop
  - Processor 1 prints “0” for A
- Ramification: store buffers can cause “strange” behavior
  - How strange depends on lots of things

```c
A=flag=0;
Processor 0
A=1;
flag=1;
Processor 1
while (!flag); // spin
print A;
```
Sequential Consistency (SC)

```
A=flag=0;
Processor 0
A=1;
flag=1;
while (!flag); // spin
print A;
```

- **Sequential consistency (SC)**
  - Formal definition of memory view programmers expect
    - Processors see their own loads and stores in program order
      + Provided naturally, even with out-of-order execution
    - But also: processors see others’ loads and stores in program order
    - And finally: all processors see same global load/store ordering
      - Last two conditions not naturally enforced by coherence
  - **Lamport definition**: multiprocessor ordering...
    - Corresponds to some sequential interleaving of uniprocessor orders
    - I.e., indistinguishable from multi-programmed uni-processor

Enforcing SC

- What does it take to enforce SC?
  - Definition: all loads/stores globally ordered
  - Translation: coherence events of all loads/stores globally ordered
  - **When do coherence events happen naturally?**
    - On cache access
    - For stores: retirement → in-order → good
      - No write buffer? Yikes, but OK with write-back D$
    - For loads: execution → out-of-order → bad
      - No out-of-order execution? Double yikes
  - Is it true that multi-processors cannot be out-of-order?
    - No, but it makes OoO a little trickier
    - Treat out-of-order loads and stores as speculative
    - Treat certain coherence events as mispeculations
      - E.g., a BW request to block with speculative load pending
Memory Consistency Models

- **Processor consistency (PC)** (x86, SPARC)
  - Allows an in-order store buffer
  - Stores can be deferred, but must be put into the cache in order
- **Release consistency (RC)** (ARM, Itanium, PowerPC)
  - Allows an un-ordered store buffer
  - Stores can be put into cache in any order

Restoring Order

- Sometimes we need ordering (mostly we don’t)
  - Prime example: ordering between “lock” and data
- How? insert **Fences (memory barriers)**
  - Special instructions, part of ISA
- Example
  - Ensure that loads/stores don’t cross lock acquire/release operation
    ```
    acquire
    fence
    critical section
    fence
    release
    ```
- How do fences work?
  - They stall execution until write buffers are empty
  - Makes lock acquisition and release slow(er)
- Use synchronization library, don’t write your own
Multiprocessors Are Here To Stay

- Moore’s law is making the multiprocessor a commodity part
  - >1B transistors on a chip, what to do with all of them?
  - Not enough ILP to justify a huge uniprocessor
  - Really big caches? $t_{hit}$ increases, diminishing $\%_{miss}$ returns

- **Chip multiprocessors (CMPs)**
  - Multiple full processors on a single chip
  - Example: IBM POWER4: two 1GHz processors, 1MB L2, L3 tags
  - Example: Sun Niagara: 8 4-way FGMT cores, 1.2GHz, 3MB L2

- Multiprocessors a huge part of computer architecture
  - Another entire course on multiprocessor architecture

Multiprocessing & Power Consumption

- Multiprocessing can be very power efficient

- Recall: dynamic voltage and frequency scaling
  - Performance vs power is NOT linear
  - Example: Intel’s Xscale
    - 1 GHz $\rightarrow$ 200 MHz reduces energy used by 30x

- Impact of parallel execution
  - What if we used 5 Xscales at 200MHz?
  - Similar performance as a 1Ghz Xscale, but **1/6th the energy**
    - 5 cores * 1/30th = 1/6th

- Assumes parallel speedup (a difficult task)
  - Remember Ahmdal’s law
# Shared Memory Summary

- Three aspects to global memory space illusion
  - **Coherence**: consistent view of individual cache lines
    - Implementation? SMP: snooping, MPP: directories
  - **Synchronization**: regulated access to shared data
    - Key feature: atomic lock acquisition operation (e.g., `t&l`)
  - **Consistency**: consistent global view of all memory locations
    - Programmers intuitively expect sequential consistency (SC)

- How do we implement this
  - Correctly
  - Cost-Effectively
  - **TAKE Compsci 221/ECE 259!!**