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"
- 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

Directory MSI Protocol

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>P0</th>
<th>P1</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: <code>%addl r1, accts, r3</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: <code>ld 0(r3), r4</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: <code>bit r4, r2, 6</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: <code>sub r4, r2, r4</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: <code>st r4, 0(r3)</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: <code>call spew_cash</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `ld` 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)
    - Bus: 2 hops (P0→P1→P0)
    - Directory: 3 hops (P0→memory→P1→P0)

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
  ➢ Atomically 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.

One Down, Two To Go

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

The Need for 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)

- What really happened?
  ➢ Access to acct[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 acct[241].bal

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)

- What really happened?
  ➢ Access to acct[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 acct[241].bal
Working Spinlock: Test-And-Set

- ISA provides an atomic lock acquisition instruction
  - Example: test-and-set
    - Atomically executes
      - ld r1,0(&lock)
      - st 1,0(&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
    - A0: t&s r1,0(&lock)
    - A1: bnez r1,A0
  - 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>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>A1: bnez r1,A0</td>
</tr>
<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>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, \( x = 0, y = 0, \text{flag} = 0, A = 0 \))
  - What value pairs can be read by the two loads? \((x, y)\) pairs:
    - thread 1: \( x \rightarrow \text{load } y \rightarrow \text{store } 1 \rightarrow x \)
    - thread 2: \( y \rightarrow \text{load } y \rightarrow \text{store } 1 \rightarrow y \)
  - What value can be read by “Load A” below?
    - thread 1: \( x \rightarrow \text{load A} \rightarrow \text{while}(\text{flag} == 0) \{ \} \)
    - thread 2: \( y \rightarrow \text{load A} \)

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
  - Eliminates stalls on write misses (mostly)
- **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!
  - 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

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
Sequential Consistency (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
- 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

Enforcing SC

- What does it take to enforce SC?
  - Definition: all loads/stores globally ordered
  - Translation: coherence events of all loads/stores globally ordered

Memory Consistency Models

- Processor consistency (PC) (x86, SPARC)
  - Allows a 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

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? \( t_{cache} \) increases, diminishing % of memory returns

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 → 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 Amdahl’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`, `s&g`)
  - **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!!**