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

Thread-Level Parallelism

- Thread-level parallelism (TLP)
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared loosely, dynamically

  - Example: database/web server (each query is a thread)
    - accts is shared, can't register allocate even if it were scalar
    - id and amt are private variables, register allocated to r1, r2
- Focus on this

Shared Memory

- Shared memory
  - Multiple execution contexts sharing a single address space
    - Multiple programs (MIMD)
    - Or more frequently: multiple copies of one program (SPMD)
  - Implicit (automatic) communication via loads and stores
    - No need for messages, communication happens naturally
      - Maybe too naturally
    - Supports irregular, dynamic communication patterns
      - Both DLP and TLP
  - Complex hardware
    - Must create a uniform view of memory
    - Several aspects to this as we will see

Admin

- Work on Projects
Shared-Memory Multiprocessors

- Provide a shared-memory abstraction
  - Familiar and efficient for programmers

Paired vs. Separate Processor/Memory?

- **Separate processor/memory**
  - Uniform memory access (UMA): equal latency to all memory
  - Simple software, doesn’t matter where you put data
    - Lower peak performance
  - Bus-based UMAs common: symmetric multi-processors (SMP)

- **Paired processor/memory**
  - Non-uniform memory access (NUMA): faster to local memory
    - More complex software: where you put data matters
    - Higher peak performance: assuming proper data placement

Shared vs. Point-to-Point Networks

- **Shared network**: e.g., bus (left) or crossbar (not shown)
  - Low latency
  - Low bandwidth: expensive to scale beyond ~16 processors
  - Shared property simplifies cache coherence protocols (later)

- **Point-to-point network**: e.g., mesh or ring (right)
  - Longer latency: may need multiple “hops” to communicate
  - Higher bandwidth: scales to 1000s of processors
  - Cache coherence protocols are more complex

Organizing Point-To-Point Networks

- **Network topology**: organization of network
  - Tradeoff performance (connectivity, latency, bandwidth) ↔ cost

- **Router chips**
  - Networks that require separate router chips are indirect
  - Networks that use processor/memory/router packages are direct
  - Distinction blurry in the multicore era

- **Point-to-point network examples**
  - Indirect tree (left)
  - Direct mesh or ring (right)

Implementation #1: Snooping Bus MP

- Bus-based systems
  - Typically small: 2–8 (maybe 16) processors
  - Typically processors split from memories (UMA)
  - Sometimes multiple processors on single chip (CMP)
  - Symmetric multiprocessors (SMPs)
  - Common, I use one everyday

- Crossbar-based systems similar, but higher B/W and cost

Implementation #2: Scalable MP

- General point-to-point network-based systems
  - Typically processor/memory/router blocks (NUMA)
    - Glueless MP: no need for additional “glue” chips
    - Can be arbitrarily large: 1000’s of processors
  - Massively parallel processors (MPPs)
  - Increasingly used for small systems
    - Eliminates need for busses, enables point-to-point wires
  - Coherent Hypertransport (AMD Opteron)
  - Intel QuickPath (Core 2)
**Issues for Shared Memory Systems**

- Three in particular
  - Cache coherence
  - Synchronization
  - Memory consistency model
  - Not unrelated to each other
- Different solutions for SMPs and MPPs
  - Will discuss SMPs only
  - CMPs? Now like SMPs, but maybe MPPs later

**An Example Execution**

- 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)

**Write-Thru Alone Doesn't Help**

- Scenario II: processors have write-thru caches
  - This time only 2 (different) copies of accts[241].bal
  - No problem? What if another withdrawal happens on processor 0?

**Hardware Cache Coherence**

- Absolute coherence
  - All copies have same data at all times
  - Hard to implement and slow
  - Not strictly necessary
- Relative coherence
  - Temporary incoherence OK (e.g., write-back)
  - As long as all loads get right values
    - i.e., no one looks at incoherent data
- Coherence controller:
  - Examines bus traffic (addresses and data)
  - Executes coherence protocol
    - What to do with local copy when you see different things happening on bus
Bus-Based Coherence Protocols

- Bus-based coherence protocols
  - Also called snooping or broadcast
  - ALL controllers see ALL transactions IN SAME ORDER
    - Bus is the ordering point
    - Protocol relies on all processors seeing a total order of requests

- Simplest protocol: write-thru cache coherence
  - Two processor-side events
    - R: read
    - W: write
  - Two bus-side events
    - BR: bus-read, read miss on another processor
    - BW: bus-write, write thru by another processor

Write-Thru Coherence Protocol

- VI (valid-invalid) protocol
  - Two states (per block)
    - V (valid): have block
    - I (invalid): don't have block
    - Can implement with valid bit
  - Protocol diagram (left)
    - Convention: event→generated-event
    - Summary
      - If anyone wants to write block
        - Give it up: transition to I state
      - Read miss gets data from memory (as normal)
    - This is an invalidate protocol
    - Simple, but wastes a lot of bandwidth
      - May be used for L1 D$

VI Protocol (Write-Back Cache)

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1,accts,r3</td>
<td>500</td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td>500</td>
</tr>
<tr>
<td>2: bhi r4,r2,r6</td>
<td>500</td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td>500</td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>500</td>
</tr>
<tr>
<td>5: call spew_cash</td>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1,accts,r3</td>
<td>0</td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td>0</td>
</tr>
<tr>
<td>2: bhi r4,r2,r6</td>
<td>0</td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td>0</td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>0</td>
</tr>
<tr>
<td>5: call spew_cash</td>
<td>0</td>
</tr>
</tbody>
</table>

VI → MSI: A realistic coherence protocol

- VI protocol is inefficient
  - Only one cached copy allowed in entire system
  - Multiple copies can't exist even if read-only
    - Not a problem in example
    - Big problem in reality
- MSI (modified-shared-invalid)
  - Fixes problem: splits "V" state into two states
    - M (modified): local dirty copy
    - S (shared): local clean copy
    - Allows either
      - Multiple read-only copies (S-state)
      - Single read/write copy (M-state)
MSI Protocol (Write-Back Cache)

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

Cache Coherence and Cache Misses

- Larger capacity: more coherence misses
  - But offset by reduction in capacity misses
- Increased block size: more coherence misses
  - False sharing: "sharing" a cache line without sharing data
  - Creates pathological "ping-pong" behavior
  - Careful data placement may help, but is difficult
- Number of processors also affects coherence misses
  - More processors: more coherence misses

Coherence Bandwidth Requirements

- How much address bus bandwidth does snooping need?
  - Well, coherence events generated on...
    - Misses (only in L2, not so bad)
    - Dirty replacements
  - Some parameters
    - 2 GHz CPUs, 2 IPC, 33% memory operations,
      - 2% of which miss in the L2, 50% of evictions are dirty
      - (0.33 * 0.02) + (0.33 * 0.02 * 0.50) = 0.01 events/insn
      - Request: 0.04 events/ns * 2 insn/cycle = 0.08 events/cycle
    - Data Response: 0.04 events/ns * 64 B/event = 2.56 GB/s
  - That's 2.5 GB/s... per processor
    - With 16 processors, that's 40 GB/s!
    - With 128 processors, that's 320 MB/s!
    - Yes, you can use multiple buses... but that hinders global ordering

More Coherence Bandwidth

- Bus bandwidth is not the only problem
  - Also processor snooping bandwidth
    - Recall: snoop implies matching address against current cache tags
      - Just a tag lookup, not data
      - 0.01 events/insn * 2 insn/cycle = 0.02 events/cycle
    - With 16 processors, each would do 0.16 tag lookups per cycle
      - If you add a port to the cache tags... OK
    - With 128 processors, each would do 1.28 tag lookups per cycle
      - If caches implement inclusion (L1 is strict subset of L2)
        - Additional snooping ports only needed on L2, still bad though
  - Upshot: bus-based coherence doesn't scale beyond 8–16

Scalable Cache Coherence

- Scalable cache coherence: two part solution
  - Part I: bus bandwidth
    - Replace non-scalable bandwidth substrate (bus)...
      - with scalable one (point-to-point network, e.g., mesh)
  - Part II: processor snooping bandwidth
    - Interesting: most snoops result in no action
      - For loosely shared data, other processors probably
      - Replace non-scalable broadcast protocol (spam everyone)...
      - with scalable directory protocol (only spam processors that care)

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 receives 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)
  - 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>PO</th>
<th>PI</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1, accts, r3</td>
<td>0: addi r1, accts, r3</td>
<td>M:400</td>
<td>M:0:500</td>
<td>(state)</td>
</tr>
<tr>
<td>1: ld 0(r3), r4</td>
<td>1: ld 0(r3), r4</td>
<td>S:500</td>
<td>S:0:500</td>
<td></td>
</tr>
<tr>
<td>2: bit r4, r2, r6</td>
<td>2: bit r4, r2, r6</td>
<td>S:400</td>
<td>S:6:400</td>
<td></td>
</tr>
<tr>
<td>3: sub r4, r2, r4</td>
<td>3: sub r4, r2, r4</td>
<td>S:500</td>
<td>S:5:1400</td>
<td></td>
</tr>
<tr>
<td>4: at r4,0(r3)</td>
<td>4: at r4,0(r3)</td>
<td>M:400</td>
<td>M:1:400</td>
<td></td>
</tr>
<tr>
<td>5: call spew_cash</td>
<td>5: call spew_cash</td>
<td>M:300</td>
<td>M:1:400</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)
    - 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

- 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
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
```

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)

```c
struct acct_t { int bal; };
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
    ```
    t&s r1,0(&lock)
    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

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

Test-and-Set Lock Correctness

```
Processor 0
AO: t&s r2,0(&lock)
AO: t&s r1,0(&lock)
CRITICAL_SECTION
AO: t&s r1,0(&lock)
AO: t&s r1,0(&lock)
AO: hres r1,80

Processor 1
A0: hres r1,80
A0: t&s r2,0(&lock)
A0: t&s r1,0(&lock)
```
+ Test-and-set lock actually works
  - Processor 1 keeps spinning

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

```
A=flag=0;

Processor 0
A0: while (!flag) : // spin
A1: print A;
A1: hres r1,80

Processor 1
A0: while (!flag) : // spin
A0: print A;
```

- 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 reorder writes

- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- Real systems act in this strange manner
Sequential Consistency (SC)

- 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

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 %$t_{miss}$ 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 → 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&s)
  - 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!!