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

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt) {
    accts[id].bal -= amt;
    spew_cash();
}
```

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
    - Simple software
      - 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
Shared-Memory Multiprocessors

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

![Memory System Diagram]

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(er) 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**
    - Fewer components, “Glueless MP”
  - Distinction blurry in the multicore era

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

![Network topology diagram](image)

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)
  - Predecessor of modern systems

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

![Implementation #1 diagram](image)

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 Interconnect (QPI)

![Implementation #2 diagram](image)

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: add 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: add 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

Processor 0
0: add 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: add 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 Incoherence

- Scenario II: processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
  - Can get incoherent (inconsistent)

Processor 0
0: add 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: add 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

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?
  - Solution: invalidate or update copies
Hardware Cache Coherence

- **Coherence Invariants**
  - Single-Writer or Multiple Reader
  - Applies to a single location (e.g., block)
  - Holds in logical, not physical time

- **Data Value Invariant**
  - Memory values propagate from writers to readers

- **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 (i.e., a processor load will read the cache)
    - **W**: write (i.e., a processor store will write the cache)
  - 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$

Coherence for Writeback caches

- **Writeback cache actions**
  - Three processor-side events
    - **R**: read
    - **W**: write
    - **WB**: write-back (select block for replacement)
  - Three bus-side events
    - **BR**: bus-read, read miss on another processor
    - **BW**: bus-write, write miss on another processor
    - **CB**: copy-back, send block back to memory or other processor

- **Point-to-point network protocols also exist**
  - Typical solution is a directory protocol
**VI (MI) Coherence Protocol**

- **VI (valid-invalid) protocol**: aka MI
  - Two states (per block)
    - V (valid): have block
      - aka M (modified) when block written
    - I (invalid): don't have block
  - Protocol summary
    - If anyone wants to read/write block
      - Give it up: transition to I state
    - copy-back on replacement or other request
    - Miss gets latest copy (memory or processor)
  - This is an invalidate protocol
  - Update protocol: copy data, don't invalidate
    - Sounds good, but wastes a lot of bandwidth
    - Hard to make writes appear atomic

**VI Protocol (Write-Back Cache)**

- **ld by processor 1 generates a BR**
  - processor 0 responds by CB its dirty copy, transitioning to S
  - st by processor 1 generates a BW
  - processor 0 responds by transitioning to I

**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) --OR--
    - Single read/write copy (M-state)

**MSI Protocol (Write-Back Cache)**

- **ld by processor 1 generates a BR**
  - processor 0 responds by CB its dirty copy, transitioning to S
  - st by processor 1 generates a BW
  - processor 0 responds by transitioning to I
Other Coherence States

- **Exclusive (E)**
  - Read misses get exclusive block if not other cache has a copy
  - Same processor can write block without another request
  - Optimizes common read-modify-write sequence

- **Owned (O)**
  - \( M \rightarrow S \) transition requires memory/lower cache update (reflection)
  - Transition from \( M \rightarrow O \) eliminates reflection
  - Cache in state O must still respond to BR and BW requests

- **Many transient states**
  - Write to I: I \( \rightarrow \) IM \( \rightarrow \) M
  - Tracks intermediate states
  - But many races: BW \( \rightarrow \) IM\(I\)D

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

Directory MSI Protocol

- 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 0
  - Dir

- Processor 1
  - Dir

- Processor 0
  - P0

- Processor 1
  - P1

- Directory
  - Dir

- \( P_0 \rightarrow P_1 \rightarrow \text{Dir} \)

- P0 sends BR to P1
- P1 sends P0 data
- P0 starts transaction with BR
- P1 starts transaction with BW
- P0 sends data to P1

- 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 0
  - Dir

- Processor 1
  - Dir

- Processor 0
  - P0

- Processor 1
  - P1

- Directory
  - Dir

- \( P_0 \rightarrow P_1 \rightarrow \text{Dir} \)

- P0 sends BR to P1
- P1 sends P0 data
- P0 starts transaction with BR
- P1 starts transaction with BW
- P0 sends data to P1
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

Coherence on Real Machines

- Many uniprocessors designed with on-chip snooping logic
  - Can be easily combined to form SMPs
    - E.g., Intel Pentium4 Xeon
  
- Larger scale (directory) systems built from smaller SMPs
  - E.g., Sun Wildfire, NUMA-Q, IBM Summit

- Some shared memory machines are not cache coherent
  - E.g., CRAY-T3D/E
  - Shared data is uncacheable
  - If you want to cache shared data, copy it to private data section
  - Basically, cache coherence implemented in software
  - Have to really know what you are doing as a programmer

Best of Both Worlds?

- Ignore processor snooping bandwidth for a minute
- Can we combine best features of snooping and directories?
  - From snooping: fast 2-hop cache-to-cache transfers
  - From directories: scalable point-to-point networks
  - In other words...

- Can we use broadcast on an unordered network?
  - Yes, and most of the time everything is fine
  - But sometimes it isn’t ... data race

- Token Coherence (TC)
  - An unordered broadcast snooping protocol ... without data races
  - Interesting, but won’t talk about here
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

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

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

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

- ISA provides an atomic lock acquisition instruction
  - Example: **test-and-set**
    ```
    tss r1,0(&lock)
    ```
  - 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: tss r1,0(&lock)
  A1: bnez r1,#A0
  ```
- More general atomic mechanisms
  - `swap`, `exchange`, `fetch-and-add`, `compare-and-swap`

Test-and-Set Lock Correctness

- Processor 0
  ```
  A0: tss r1,0(&lock)
  A1: bnez r1,#A0
  ```
- Processor 1
  ```
  CRITICAL_SECTION
  A0: tss r1,0(&lock)
  A1: bnez r1,#A0
  ```
- Processor 0 keeps spinning

Test-and-Set Lock Performance

- But performs poorly in doing so
  - Consider 3 processors rather than 2
  - Processor 0 (not shown) has the lock and is in the critical section
  - But what are processors 1 and 2 doing in the meantime?
    - Loops of `tss`, each of which includes a `st`
      - Taking turns invalidating each others cache lines
      - Generating a ton of useless bus (network) traffic

Test-and-Test-and-Set Locks

- Solution: **test-and-test-and-set locks**
  - New acquire sequence
    ```
    A0: ld r1,0(&lock)
    A1: tss r1,0(&lock)
    ```
  - Within each loop iteration, before doing a `tss`
    - Spin doing a simple test (14) to see if lock value has changed
    - Only do a `tss (st)` if lock is actually free
  - Processors can spin on a busy lock locally (in their own cache)
  - Less unnecessary bus traffic
Test-and-Test-and-Set Lock Performance

- Processor 0 releases lock, informs (invalidates) processors 1 and 2
- Processors 1 and 2 race to acquire, processor 1 wins

Processor 1

A0: ld r1,0(&lock)  
A1: bnez r1,A0  
A0: ld r1,0(&lock)  
// lock released by processor 0
A0: ld r1,0(&lock)  
A1: bnez r1,A0  
A2: addi r1,1,r1  
A3: t&s r1,(&lock)  
A4: bnez r1,A0  
CRITICAL_SECTION

Processor 2

A0: ld r1,0(&lock)  
A1: bnez r1,A0  
A1: bnez r1,A0  
A0: ld r1,0(&lock)  
A0: ld r1,0(&lock)  
A1: bnez r1,A0  
A0: ld r1,0(&lock)  
A1: bnez r1,A0  

S:1  S:1  S:1
I:  I:  I:
S:1  S:1  S:1
S:1  S:1  S:1
S:1  S:1  S:1

Queue Lock Performance

- Processor 0 releases lock, informs only processor 1

Queue Locks

- Test-and-test-and-set locks can still perform poorly
  - If lock is contended for by many processors
  - Lock release by one processor, creates “free-for-all” by others
    - Network gets swamped with t&s requests

- Queue lock
  - When lock is released by one processor...
  - Directory doesn’t notify (by invalidations) all waiting processors
  - Instead, chooses one and sends invalidation only to it
  - Others continue spinning locally, unaware lock was released
  - Effectively, directory passes lock from one processor to the next
  - Greatly reduced network traffic

A Final Word on Locking

- A single lock for the whole array may restrict parallelism
- Will force updates to different accounts to proceed serially
- Solution: one lock per account
- Locking granularity: how much data does a lock lock?
- A software issue, but one you need to be aware of

struct acct_t { int bal, lock; };  
shared struct acct_t accts[MAX_ACCT];  
int id, amt;  
acquire(accts[id].lock);  
if (accts[id].bal >= amt) {  
  accts[id].bal -= amt;  
  spew_cash(); }  
release(accts[id].lock);  

A0: ld r1,0(&lock)  
A1: bnez r1,A0  
A0: ld r1,0(&lock)  
A0: ld r1,0(&lock)  
A1: bnez r1,A0  
A0: ld r1,0(&lock)  
A1: bnez r1,A0  

S:1  S:1  S:1
S:1  S:1  S:1
S:1  S:1  S:1
S:1  S:1  S:1
S:1  S:1  S:1

CS/ECE 752 (Wood): Shared-Memory Multiprocessors 41
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

Sequential Consistency (SC)

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

- **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!
    - 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
SC + OOO

- Recall: opportunistic load scheduling in a uni-processor
  - Loads issue speculatively relative to older stores
    - Stores scan for younger loads to same address have issued
    - Find one? Ordering violation → flush and restart
    - In-flight loads effectively “snoop” older stores from same process

- SC + OOO can be reconciled using same technique
  - Write bus requests from other processors snoop in-flight loads
  - Think of ROB as extension of the cache hierarchy
  - MIPS R10K does this

- SC implementable, but overheads still remain:
  - Write buffer issues
  - Complicated ld/st logic

Is SC Really Necessary?

- SC
  + Most closely matches programmer’s intuition (don’t under-estimate)
    - Restricts optimization by compiler, CPU, memory system
  - Supported by MIPS, HP PA-RISC

- Is full-blown SC really necessary? What about...
  - All processors see same total order
  - Loads must respect program order
  - Store must respect program order
  - But loads can move ahead of stores
  + Allows processors to have in-order write buffers
    - Doesn’t confuse programmers too much
  - Total Store Ordering (TSO): e.g., Intel IA-32, SPARC

Weak Memory Ordering

- For properly synchronized programs
  - Only acquire/releases must be strictly ordered

- Why? Acquire-release pairs define critical sections
  - Between critical-sections: data is private
    - Globally unordered access OK
  - Within critical-section: access to shared data is exclusive
    - Globally unordered access also OK

  - Implication: compiler or dynamic scheduling is OK
    - As long as re-orderings do not cross synchronization points

- Weak Ordering (WO): Alpha, IA-64, PowerPC
  - ISA provides fence insns to indicate scheduling barriers
    - Proper use of fences is somewhat subtle
  - Use synchronization library, don’t write your own

SC for DRF

- Sequential Consistency for Data Race Free programs
  - Basis of C++ and Java memory models
  - Originally defined by Adve and Hill
  - Programmers/Compilers must identify synchronization
    - Use acquire and release to synchronize

  - Hardware can reorder memory operations within critical sections
    - If a tree falls in the woods, and there is no one there to hear it, does it make a sound?
    - If two memory operations are reordered, but it requires a data race to detect it, does it violate SC for DRF?
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? $t_{miss}$ 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 PGMT 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 CS/ECE 757!!**
### A Protocol Optimization

#### Cache-to-cache transfers (CCT)
- If data you need is in both memory and other cache...
- Better to get it from the other cache
- SRAM is faster than DRAM
- Especially true if cache block is dirty
- Otherwise, writeback followed by memory read
- If multiple blocks have copies, who does CCT?
  - One cache designated as "owner"

### Another Protocol Optimization

- Most modern protocols also include **E (exclusive)** state
  - Interpretation: can write to this block, but haven't yet
  - Why is this state useful?

### Cache Coherence and Cache Misses

- A coherence protocol can affect a cache's miss rate ($\%_{\text{miss}}$)
  - Requests from other processors can invalidate (evict) local blocks
  - 4C miss model: compulsory, capacity, conflict, **coherence**
  - **Coherence miss**: miss to a block evicted by bus event
    - As opposed to a processor event
    - Example: direct-mapped 16B cache, 4B blocks, nibble notation

<table>
<thead>
<tr>
<th>Cache contents (state:address)</th>
<th>Event</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>5:0000, M:0010, S:0030</td>
<td>Rd/0030</td>
<td>Conflict Miss</td>
</tr>
<tr>
<td>5:0000, M:0010, S:0020, M:0030</td>
<td>Wr/0030</td>
<td>Upgrade Miss</td>
</tr>
<tr>
<td>5:0000, M:0010, S:0020, M:0030</td>
<td>BusInv:0001</td>
<td>Nothing</td>
</tr>
<tr>
<td>5:0000, M:0010, S:0020, M:0030</td>
<td>BusInv:0002</td>
<td>Invalidation</td>
</tr>
<tr>
<td>5:0000, M:0010, S:0020</td>
<td>Rd/0030</td>
<td>Compulsory Miss</td>
</tr>
<tr>
<td>5:0000, M:0010, S:0020, S:0303</td>
<td>Rd/0030</td>
<td>Coherence Miss</td>
</tr>
<tr>
<td>5:0000, M:0010, S:0020, S:0303</td>
<td>Rd/0030</td>
<td>Conflict Miss</td>
</tr>
</tbody>
</table>

### Cache Coherence and Cache Misses

- Cache parameters interact with coherence 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?
  - 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
  - \((0.33 \times 0.02) + (0.33 \times 0.02 \times 0.50)\) = 0.01 events/insn
  - 0.01 events/insn * 2 insn/cycle * 2 cycle/ns = 0.04 events/ns
  - 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 GB/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 = 0.01 events/cycle per processor
  - With 16 processors, each would do 0.16 tag lookups per cycle
  - 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 bandwidth 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)

Spin Lock Strawman (Does not work)

- Spin lock: software lock implementation
  - acquire(lock) : while (lock != 0) : lock = 1;
  - "Spin" while lock is 1, wait for it to turn 0

    A0:  ld 0(&lock),r6
    A1:  bnez r6,A0
    A2:  addi r6,1,r6
    A3:  st r6,0(&lock)

  - release(lock) : lock = 0;

    R0:  st r0,0(&lock)  // r0 holds 0
Spin Lock Strawman (Does not work)

- Spin lock makes intuitive sense, but doesn’t actually work
  - Loads/stores of two acquire sequences can be interleaved
  - Lock acquire sequence also not atomic
  - Definition of “squeezing toothpaste”
  - Note, release is trivially atomic

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0: ld 0(&amp;lock),r6</td>
<td>A0: ld r6,0(&amp;lock)</td>
</tr>
<tr>
<td>A1: bnez r6,$A0</td>
<td>A1: bnez r6,$A0</td>
</tr>
<tr>
<td>A2: addi r6,1,r6</td>
<td>A2: addi r6,1,r6</td>
</tr>
<tr>
<td>A3: st r6,0(&amp;lock)</td>
<td>A3: st r6,0(&amp;lock)</td>
</tr>
<tr>
<td>CRITICAL_SECTION</td>
<td>CRITICAL_SECTION</td>
</tr>
</tbody>
</table>

Better Implementation: SYSCALL Lock

- Implement lock in a SYSCALL
  - Kernel can control interleaving by disabling interrupts
  - Works...
  - But only in a multi-programmed uni-processor
  - Hugely expensive in the common case, lock is free

```
ACQUIRE_LOCK:
A0: enable_interrupts
A1: disable_interrupts
A2: ld r6,0(&lock)
A3: bnez r6,$A0
A4: addi r6,1,r6
A5: st r6,0(&lock)
A6: enable_interrupts
A7: jr $r31
```

Shared Memory Summary

- Shared-memory multiprocessors
  - Simple software: easy data sharing, handles both DLP and TLP
  - Complex hardware: must provide illusion of global address space
- Two basic implementations
  - Symmetric (UMA) multi-processors (SMPs)
    - Underlying communication network: bus (ordered)
      + Low-latency, simple protocols that rely on global order
        - Low-bandwidth, poor scalability
  - Scalable (NUMA) multi-processors (MPPs)
    - Underlying communication network: point-to-point (unordered)
      + Scalable bandwidth
      - Higher-latency, complex protocols

SC + OOO vs. WO

- Big debate these days
  - Is SC + OOO equal to WO performance wise?
  - And if so, which is preferred?
- Another hot button issue
  - Can OOO be used to effectively speculate around locks?
  - Short answer: yes