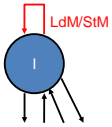


"Scalable" Cache Coherence



Part I: bus bandwidth

Replace non-scalable bandwidth substrate (bus)...
...with scalable one (point-to-point network, *e.g.*, mesh)

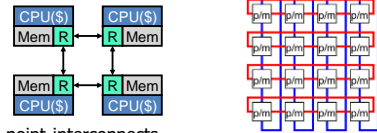
Part II: processor snooping bandwidth

- Most snoops result in no action
- Replace non-scalable broadcast protocol (spam everyone)...
...with scalable **directory protocol** (only notify processors that care)

57

57

Scalable Cache Coherence



Point-to-point interconnects

- **Glueless MP**: no need for additional "glue" chips
 - + Can be arbitrarily large: 1000's of processors
 - **Massively parallel processors (MPPs)**
 - Only government (DoD) has cache-coherent MPPs...
 - Companies have much smaller systems: 32-64 processors
 - **Scalable multi-processors**
- AMD Opteron/Phenom – point-to-point, glueless, broadcast
- Distributed memory: non-uniform memory architecture (NUMA)
- Multicore: on-chip mesh interconnection networks

58

58

Directory Coherence Protocols

Observe: 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

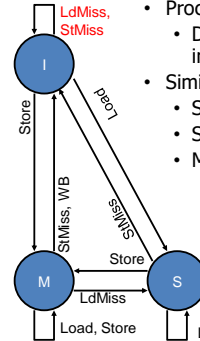
Directories: non-broadcast coherence protocol

- 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
- For multicore w/ shared L3, put directory info in cache tags

59

59

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 **red** arcs/actions
 - Events that would not trigger action anyway
 - + Directory won't bother you unless you need to act

60

60

MSI Directory Protocol

Processor 0	Processor 1	P0	P1	Directory
0: addi r1,accts→r3				--:~500
1: ld 0(r3),r4		S:500		S:0:500
2: blt r4,r2,done				
3: sub r4,r2→r4		M:400		M:0:500 (stale)
4: st r4,0(r3)				
	0: addi r1,accts→r3			
	1: ld 0(r3),r4	S:400	S:400	S:0:1:400
	2: blt r4,r2,done			
	3: sub r4,r2→r4			
	4: st r4,0(r3)	I:	M:300	M:1:400

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

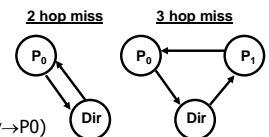
- Directory sends BW to P0, P0 goes to **I**

61

61

Directory Flip Side: Latency

- Directory protocols
 - + Lower bandwidth consumption → more scalable
 - Longer latencies



- Two read miss situations

- Unshared: get data from memory

- Snooping: 2 hops (P0→memory→P0)
- Directory: 2 hops (P0→memory→P0)

- Shared or exclusive: get data from other processor (P1)

- Assume cache-to-cache transfer optimization

- Snooping: 2 hops (P0→P1→P0)

- Directory: **3 hops** (P0→memory→P1→P0)

- Common, many processors → high probability someone has it

62

62

Directory Flip Side: Complexity

- Latency is not the only issue for directories
 - Subtle correctness issues as well
 - Stem from unordered nature of underlying inter-connect
- Individual requests to single cache must be ordered
 - Bus-based snooping: all processors see all requests in same order
 - Ordering automatic
 - Point-to-point network: requests may arrive in different orders
 - Directory has to enforce ordering explicitly
 - Cannot initiate actions on request B...
 - ..until all relevant processors complete actions on request A
 - Requires directory to collect acks, queue requests, etc.
- Directory protocols
 - Obvious in principle
 - Complicated in practice

63

63

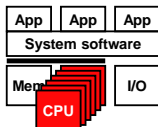
Coherence on Real Machines

- Many uniprocessors designed with on-chip snooping logic
 - Can be easily combined to form multi-processors
 - e.g., Intel Pentium4 Xeon
 - And multicore, of course
- Larger scale (directory) systems built from smaller MPs
 - e.g., Sun Wildfire, NUMA-Q, IBM Summit
- Some shared memory machines are **not cache coherent**
 - e.g., CRAY-T3D/E
 - Shared data is uncachable
 - 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

64

64

Roadmap Checkpoint



- Thread-level parallelism (TLP)
- Shared memory model
 - Multiplexed uniprocessor
 - Hardware multithreading
 - Multiprocessing
- Synchronization
 - Lock implementation
 - Locking gotchas
- Cache coherence
 - Bus-based protocols
 - Directory protocols
- **Memory consistency models**

65

65

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

- (0,0) and (1,1) easy to see
- load x, store 1 → y, load y, store 1 → x gives (0,1)
- Is it possible to get (1,0)?

66

66

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) { }
store 1 → flag	load A

67

67

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

68

68

Hiding Store Miss Latency

- Recall (back from caching unit)
 - Hiding store miss latency
 - How? Store buffer
- Said it would complicate multiprocessors
 - Yes, it does!

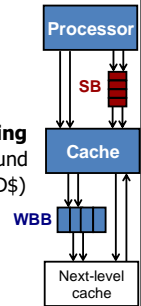
69

69

Write Misses and Store Buffers

- Read miss?
- Load can't go on without the data → must stall
- Write miss?
- Technically, no one needs data → why stall?

- Store buffer:** a small buffer
- Stores put addr/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



70

70

Store Buffers & Consistency

```

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

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

71

71

Coherence vs. Consistency

```

A=0  flag=0
Processor 0      Processor 1
A=1;             while (!flag); // spin
flag=1;          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?
 - P0 has a coalescing store buffer that reorders writes**
 - Or out-of-order execution**
 - Or compiler re-orders instructions**
- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- Real systems** act in this strange manner
 - What is allowed is defined as part of the ISA of the processor

72

72

Memory Consistency Models

- Sequential consistency (SC)** (MIPS, PA-RISC)
 - 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
 - Corresponds to some sequential interleaving of uniprocessor orders
 - Indistinguishable from multi-programmed uni-processor**
- 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

73

73

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**

74

74

Shared Memory Summary

- **Synchronization:** regulated access to shared data
 - Key feature: atomic lock acquisition operation (e.g., `t&&=s`)
 - Performance optimizations: test-and-test-and-set, queue locks
- **Coherence:** consistent view of individual cache lines
 - Absolute coherence not needed, relative coherence OK
 - VI and MSI protocols, cache-to-cache transfer optimization
 - Implementation? snooping, directories
- **Consistency:** consistent view of all memory locations
 - Programmers intuitively expect sequential consistency (SC)
 - Global interleaving of individual processor access streams
 - Not always naturally provided, may prevent optimizations
 - Weaker ordering: consistency only for synchronization points

75

75

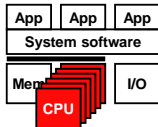
Flynn's Taxonomy

- Proposed by Michael Flynn in 1966
- SISD – single instruction, single data
 - Traditional uniprocessor
- SIMD – single instruction, multiple data
 - Execute the same instruction on many data elements
 - Vector machines, graphics engines
- MIMD – multiple instruction, multiple data
 - Each processor executes its own instructions
 - Multicores are all built this way
 - SPMD – single program, multiple data (extension proposed by Frederica Darema)
 - MIMD machine, each node is executing the same code
- MISD – multiple instruction, single data
 - Systolic array

76

76

Summary



- Thread-level parallelism (TLP)
- Shared memory model
 - Multiplexed uniprocessor
 - Hardware multithreading
 - Multiprocessing
- Synchronization
 - Lock implementation
 - Locking gotchas
- Cache coherence
 - Bus-based protocols
 - Directory protocols
- Memory consistency models

77

77