Lecture 7: Implementing Cache Coherence

- Topics: implementation details

Implementing Coherence Protocols

- Correctness and performance are not the only metrics
- Deadlock: a cycle of resource dependencies, where each process holds shared resources in a non-preemptible fashion
- Livelock: similar to deadlock, but transactions continue in the system without each process making forward progress
- Starvation: an extreme case of unfairness

Basic Implementation

- Assume single level of cache, atomic bus transactions

- It is simpler to implement a processor-side cache controller that monitors requests from the processor and a bus-side cache controller that services the bus

- Both controllers are constantly trying to read tags
  - tags can be duplicated (moderate area overhead)
  - unlike data, tags are rarely updated
  - tag updates stall the other controller

Reporting Snoop Results

- Uniprocessor system: initiator places address on bus, all devices monitor address, one device acks by raising a wired-OR signal, data is transferred

- In a multiprocessor, memory has to wait for the snoop result before it chooses to respond – need 3 wired-OR signals: (i) indicates that a cache has a copy, (ii) indicates that a cache has a modified copy, (iii) indicates that the snoop has not completed

- Ensuring timely snoops: the time to respond could be fixed or variable (with the third wired-OR signal), or the memory could track if a cache has a block in M state

Non-Atomic State Transitions

- Note that a cache controller's actions are not all atomic: tag look-up, bus arbitration, bus transaction, data/tag update

- Consider this: block A in shared state in P1 and P2; both issue a write; the bus controllers are ready to issue an upgrade request and try to acquire the bus; is there a problem?

- The controller can keep track of additional intermediate states so it can react to bus traffic (e.g. S→M, I→M, I→S,E)

- Alternatively, eliminate upgrade request; use the shared wire to suppress memory's response to an exclusive-rd

Serialization

- Write serialization is an important requirement for coherence and sequential consistency – writes must be seen by all processors in the same order

- On a write, the processor hands the request to the cache controller and some time elapses before the bus transaction happens (the external world sees the write)

- If the writing processor continues its execution after handing the write to the controller, the same write order may not be seen by all processors – hence, the processor is not allowed to continue unless the write has completed

Livelock

- Livelock can happen if the processor-cache handshake is not designed correctly

- Before the processor can attempt the write, it must acquire the block in exclusive state

- If all processors are writing to the same block, one of them acquires the block first – if another exclusive request is seen on the bus, the cache controller must wait for the processor to complete the write before releasing the block -- else, the processor’s write will fail again because the block would be in invalid state

Atomic Instructions

- A test&set instruction acquires the block in exclusive state and does not release the block until the read and write have completed

- Should an LL bring the block in exclusive state to avoid bus traffic during the SC?

- Note that for the SC to succeed, a bit associated with the cache block must be set (the bit is reset when a write to that block is observed or when the block is evicted)

- What happens if an instruction between the LL and SC causes the LL-SC block to always be replaced?

Multilevel Cache Hierarchies

- Ideally, the snooping protocol employed for L2 must be duplicated for L1 – redundant work because of blocks common to L1 and L2

- Inclusion greatly simplifies the implementation

Maintaining Inclusion

- Assuming equal block size, if L1 is 8KB 2-way and L2 is 256KB 8-way, is the hierarchy inclusive? (assume that an L1 miss brings a block into L1 and L2)

- Assuming equal block size, if L1 is 8KB direct-mapped and L2 is 256KB 8-way, is the hierarchy inclusive?

- To maintain inclusion, L2 replacements must also evict relevant blocks in L1

Intra-Hierarchy Protocol

- Some coherence traffic needs to be propagated to L1; likewise, L1 write traffic needs to be propagated to L2
- What is the best way to implement the above? More traffic? More state?
- In general, external requests propagate upward from L3 to L1 and processor requests percolate down from L1 to L3
- Dual tags are not as important as the L2 can filter out bus transactions and the L1 can filter out processor requests

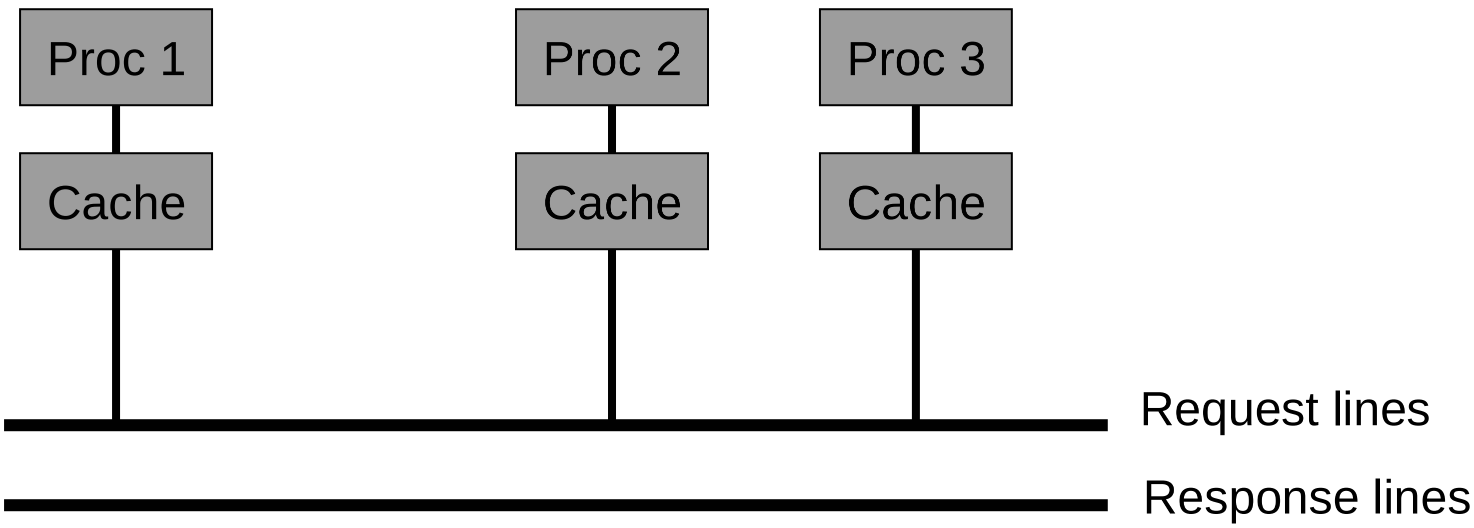
Split Transaction Bus

- What would it take to implement the protocol correctly while assuming a split transaction bus?

- Split transaction bus: a cache puts out a request, releases the bus (so others can use the bus), receives its response much later

- Assumptions:
  - only one request per block can be outstanding
  - separate lines for addr (request) and data (response)

Split Transaction Bus



Design Issues

- When does the snoop complete? What if the snoop takes a long time?

- What if the buffer in a processor/memory is full? When does the buffer release an entry? Are the buffers identical?

- How does each processor ensure that a block does not have multiple outstanding requests?

- What determines the write order – requests or responses?

Design Issues II

- What happens if a processor is arbitrating for the bus and witnesses another bus transaction for the same address?
- If the processor issues a read miss and there is already a matching read in the request table, can we reduce bus traffic?

Shared Cache Designs

- There are benefits to sharing the first level cache among many processors (for example, in a CMP):
  - no coherence protocol
  - low cost communication between processors
  - better prefetching by processors
  - working set overlap allows shared cache size to be smaller than combined size of private caches
  - improves utilization
- Disadvantages:
  - high contention for ports
  - longer hit latency (size and proximity)
  - more conflict misses

TLBs

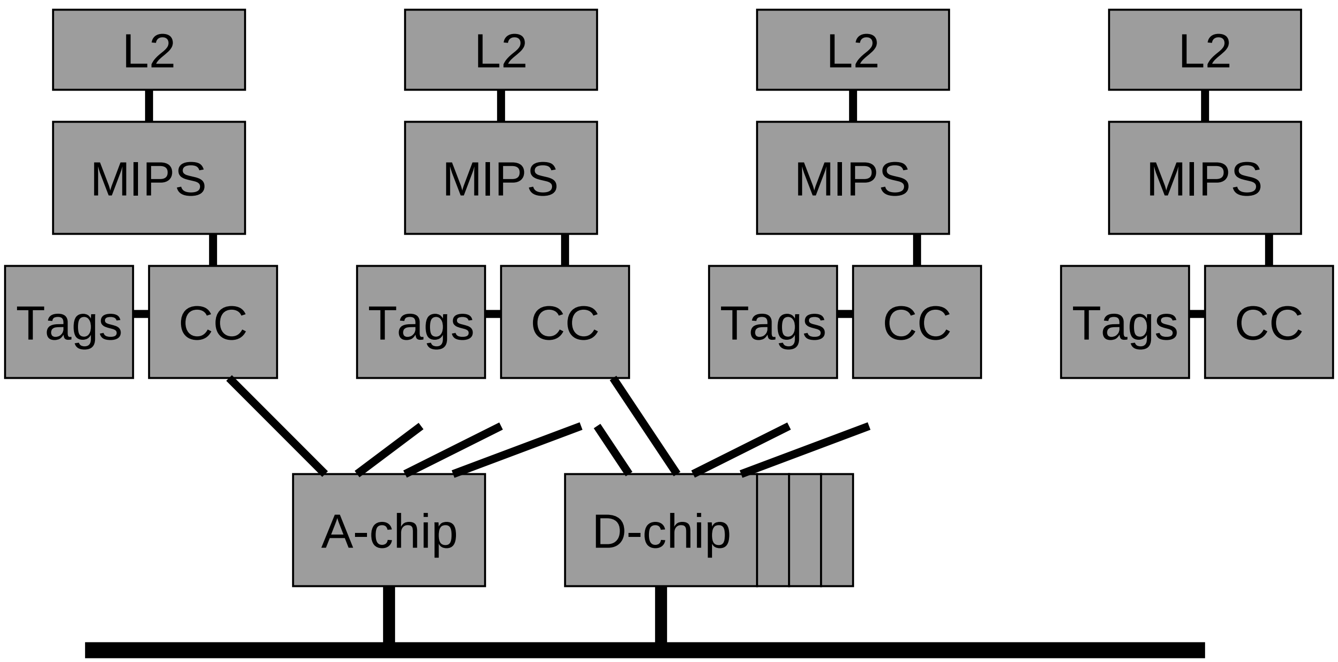
- Recall that a TLB caches virtual to physical page translations
- While swapping a page out, can we have a problem in a multiprocessor system?
- All matching entries in every processor's TLB must be removed
- TLB shootdown: the initiating processor sends a special instruction to other TLBs asking them to invalidate a page table entry

Case Study: SGI Challenge

- Supports 18 or 36 MIPS processors
- Employs a 1.2 GB/s 47.6 MHz system bus (Powerpath-2)
- The bus has 256-bit-wide data, 40-bit-wide address, plus 33 other signals (non multiplexed)
- Split transaction, supporting eight outstanding requests
- Employs the MESI protocol by default – also supports update transactions

Processor Board

- Each board has four processors (to reduce the number of slots on the bus from 36 to 9)
- A-chip has request tables, arbitration logic, etc.



Latencies

- 75ns for an L2 cache hit
- 300ns for a cache miss to percolate down to the A-chip
- Additional 400ns for the data to be delivered to the D-chips across the bus (includes 250ns memory latency)
- Another 300ns for the data to reach the processor
- Note that the system bus can accommodate 256 bits of data, while the CC-chip to processor interface can handle 64 bits at a time

Sun Enterprise 6000

- Supports 30 UltraSparcs
- 2.67 GB/s 83.5 MHz Gigaplane system bus
- Non multiplexed bus with 256 bits of data, 41 bits of address, and 91 bits of control/error correction, etc.
- Split transaction bus with up to 112 outstanding requests
- Each node speculatively drives the bus (in parallel with arbitration)
- L2 hits are 40 ns, memory access is 300 ns

Title

- Bullet