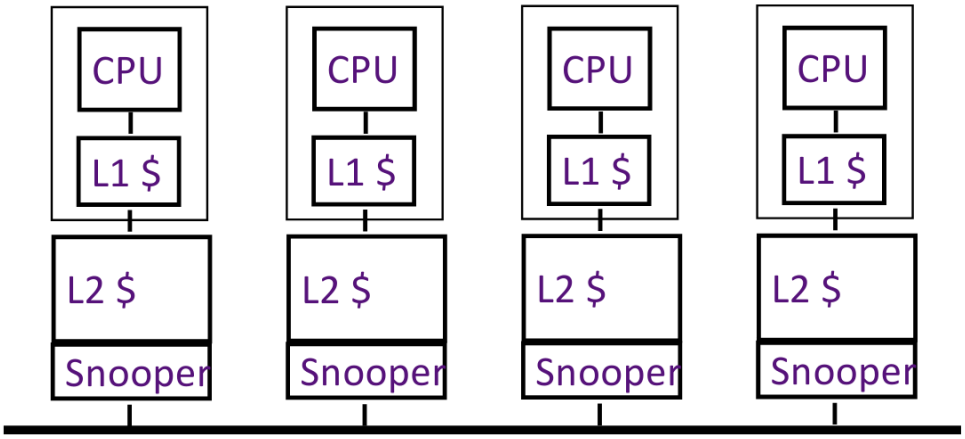
CS152 Section 13

Q1: Cache Coherence

Q1.1: Inclusion Policy (2020 Q4 A)

In lecture, it was mentioned that an inclusive L2 cache can act as a filter to reduce the amount of L1 coherence traffic in a snoopy cache-coherence protocol. If a coherence request misses in the L2 cache, there is no need to probe the L1 cache for the given line.



Explain how a strictly exclusive L2 cache can also be used to optimize snooping.

Could a non-inclusive, non-exclusive L2 cache (i.e., neither strictly inclusive nor strictly exclusive) be similarly used to optimize snooping? Explain.

Q1.2: False Sharing (2019 Q4 D)

In the table below, indicate which memory operations experience a hit, true sharing miss, or false sharing miss under an MSI protocol, and explain why. Assume x1 and x2 reside in the same cache line, and both words are read by both processors P1 and P2 before this sequence.

Time	P1	P2	Hit, True Sharing Miss, False Sharing Miss?
1		write x2	True miss: invalidate x1 in P1
2	read x1		
3		read x1	
4	write x2		
5		write x1	

Q1.3: Directory-Based Coherence (2019 Q4 E-F)

Consider a simplified version of the directory-based cache coherence scheme from PS5 (Handout #6). Our system consists of 16 cores, each with write-back, write-allocate private caches. All caches are connected to a single directory. Specifically, we are interested in the series of coherence events that transpire to service a core's load or store under different initial conditions. For example, consider a load miss from core 0 to a line is currently not cached by any core in the system:

Time	Agent	Current State	Event / Message Received	Next State	Messages Sent
0	Core 0	C-nothing	Load	C-pending	ShReq(0)
1	Directory	R(ε) (empty)	ShReq(0)	R(0)	ShRep(0)
2	Core 1	C-pending	ShRep(0)	C-shared	N/A

Simplifications:

- Assume agents can send and receive multiple messages simultaneously
- Assume messages take one time step to reach their destinations
- If a set of events may happen in parallel, indicate this by setting the "time" field to the same value
- Include only the ID field of messages (i.e., neglect data, address fields)

In the table below, indicate the series of events that occur to service a load miss from core 0 when core 4 has the line cached in the C-exclusive state.

Time	Agent	Current State	Event / Message Received	Next State	Messages Sent
0	Core 0	C-nothing	Load	C-pending	ShReq(0)

In the table below, indicate the series of events that occur to service a store miss from core 0 when cores 0, 4 and 8 have the line cached in the C-shared state.

Time	Agent	Current State	Event / Message Received	Next State	Messages Sent
0	Core 0	C-shared	Store	C-pending	ExReq(0)

Q1.4: Directory-based Coherence (2020 Q4 C)

Consider the directory-based coherence protocol described in Handout #6 from Problem Set 5. Assume that message passing maintains FIFO order: All messages between the same source and destination are always received in the same order that they were sent. Also assume that each site has sufficient queuing capacity to buffer all incoming messages without drops.

Q1.4a

Consider the situation where a cache is sent an InvReq message for a given cache line. This occurs only if the directory state indicates that the site is a current sharer of the memory block, and the directory intends to invalidate the copy in the cache before granting exclusive access to another cache.

Typically, one expects the line to be in the C-shared state when the InvReq arrives. How is it also possible for the cache to receive the InvReq message while it has the line in the C-nothing state (row #5 in Table H12-1 of Handout #6) – in other words, when the line is not actually present? Why does ignoring InvReq work out correctly in this case?

Q1.4b

Consider the case where a cache requests a line in the C-exclusive state (ExReq) when the line is clean and shared by other caches. To reduce the response latency, it is a tempting idea to send the ExRep message with the data to the requestor in parallel to sending InvReq to the other caches. Does this optimization work correctly?

Q3: Load/Store Units and Memory Consistency (2018 Q3)

Table 2.1 shows the current state of the store queue in an out-of-order core of a chip multiprocessor (CMP). Stores are kept in the store queue until they commit. The instruction number indicates the order of instructions in the program, with lower numbers being earlier in program order.

Table 2.2 shows the values stored in the non-blocking data cache. Loads following cache misses can read the data from the data cache if the memory model is not violated. Caches are coherent across processors.

Table 2.3, Table 2.4, Table 2.5, and Table 2.6 show the current state of the load queue. Assume that all stores and loads are for the full 32-bit word and aligned to 32 bits. The processor uses conservative out-of-order load/store execution.

Table 2.1: Store Queue

Instruction Number	Address	Value
3	0x1000	0xF00D3ABC
6	0x2000	Unknown
11	Unknown	Unknown
15	0x3000	0xDEADBEEF
17	Unknown	Unknown

Table 2.2:Data Cache

Valid?	Address	Value
Y	0x1000	0xAACCBDAF
Y	0x2000	0xBADE2140
Y	0x3000	0x1234ABCD
N	0x4000	Unknown

Q3.1: Sequential Consistency

Under **sequential consistency (SC)**, assuming the stores make no progress, can each load in Table 2.3 complete? If so, what value does it read?

Table 2.3: Load Queue

Instruction Number	Address	Can Complete?	Value
1	0x2000		
5	0x3000		
7	0x1000		
8	0x4000		
9	0x2000		
16	0x3000		
18	0x3000		

Q3.2: Total Store Ordering

Under **total store ordering (TSO)**, assuming the stores make no progress, can each load in Table 2.4 complete? If so, what value does it read?

Table 2.4: Load Queue

Instruction Number	Address	Can Complete?	Value
1	0x2000		
5	0x3000		
7	0x1000		
8	0x4000		
9	0x2000		
16	0x3000		
18	0x3000		

Q3.3: Weak Ordering

Under the **weak multi-copy-atomic memory model**, assuming the stores make no progress, can each load in Table 2.5 complete? If so, what value does it read?

Table 2.5: Load Queue

Instruction Number	Address	Can Complete?	Value
1	0x2000		
5	0x3000		
7	0x1000		
8	0x4000		
9	0x2000		
16	0x3000		
18	0x3000		

Q3.4: Multithreading

Now, simultaneous multithreading (SMT) is implemented in each out-of-order core to support multithreading in a single processor. Assume all stores in Table 2.1 have been issued by Thread 1 while all loads in Table 2.6 are issued by Thread 2. Assume the stores make no progress. Under the **weak multi-copy-atomic memory model**, can each load in Table 2.6 complete? If so, what value does it read?

Table 2.6: Load Queue in Thread 2

Instruction Number	Address	Can Complete?	Value
1	0x2000		
5	0x3000		
7	0x1000		
8	0x4000		
9	0x2000		
16	0x3000		
18	0x3000		

Q3: Synchronization (2018 Q6)

In the following question, you will implement semaphores in C using various synchronization primitives. A semaphore is a data structure used to control access to a critical section so that only N threads can enter the critical section at a time.

The semaphore is a shared integer initialized to N. Two functions, “wait” and “signal”, must be implemented. The wait function spins until the integer is >0. It then decrements the integer and finishes. The “signal” function increments the integer. In this way, only N threads can pass through the “wait” function before the “signal” function is called.

```
// Example non-atomic implementations of wait and signal
```

```
void wait(int *sem) {
    while (*sem <= 0) {}
    *sem = *sem - 1;
}
```

```
void signal(int *sem) {
    *sem = *sem + 1;
}
```

Q3.1: AMO

The following code attempts to implement wait and signal using atomic add operations.

```
// Atomically reads the value of *dst, adds inc to it,
// and writes the value back
void atomic_add(int *dst, int inc);
```

```
void wait(int *sem) {
    while (*sem <= 0) {}
    atomic_add(sem, -1);
}
```

```
void signal(int *sem) {
    atomic_add(sem, 1);
}
```

Assuming the memory system is sequentially consistent, does this code correctly implement wait and signal? If not, how might this implementation fail to guarantee that only N threads enter the critical section?

Q3.2: CAS

Implement the wait and signal functions using a compare-and-swap instruction.

```
// Atomically checks if (*dst == old)
// Writes new to *dst if they match
// Returns 1 on success and 0 on failure
int compare_and_swap(int *dst, int old, int new);
```

```
void wait(int *sem) {
    int success, old;
    do {









    } while (!success);
}
```

```
void signal(int *sem) {
    int success, old;
    do {









    } while (!success);
}
```

Q3.3: LR/SC

Implement the wait and signal functions using load-reserved and store-conditional

```
// Load and return the value from *src and set a reservation
int load_reserved(int *src);

// Store new to *dst if the reservation is still set
// Return 1 on success and 0 on failure
int store_conditional(int *dst, int new);
```

```
void wait(int *sem) {
    int success, old;
    do {









    } while (!success);
}
```

```
void signal(int *sem) {
    int success, old;
    do {









    } while (!success);
}
```