1. Describe the optimization that MESI does to reduce memory transactions.
Ans.: The MESI protocol reduces memory transactions by adding the Exclusive (E) state to the MSI protocol. In MESI, a cache line in the Exclusive state, being the only valid copy, allows writes without invalidation broadcasts. This avoids the unnecessary coherence traffic seen in MSI, where writes to Shared lines always trigger invalidation even when other caches do not possess the data. MESI ensures consistency with reduced communication.

2. Describe the optimization that MOESI does to reduce memory and bus Transactions.
Ans.: MOESI optimizes memory and bus usage over MESI by adding the Owned (O) state, allowing modified data to be shared without immediate memory write-backs. Unlike MESI, where the Modified (M) state requires a write-back when transitioning to Shared (S), MOESI allows a cache line in the Owned state to serve as the source of truth, directly fulfilling requests. This reduces memory accesses and bus transactions, improving coherence efficiency.

3. Describe high-level details of your implementations.
Ans.: a) MESI
1) For the MESI_processor_access() function, I first check if a cache line already exists for the address being accessed. In the scenario where the line doesn’t exist, (this represents a cold-miss scenario) I create a cache line using the provided fillLine API and set its state to INVALID. This initial check serves the purpose of ensuring that a NULL value is not returned when I try to check the state of the cache line. I am not counting this invalidation towards the invalidation statistic as it is specific to my implementation.
2) Next, I first record the state of the address in the processor’s cache. Based on this state & the r/w nature of the access, I make the following decisions:
i) If the operation is a read, I increment the ‘read’ statistic. If the state is INVALID, I initiate the BusRd snoop and increment the ReadMiss stat. Based on the function parameter value ‘copy’, I change the state to SHARED or EXCLUSIVE and increase the execution times accordingly. If the state is not INVALID, I record a readHit and increase the execution time accordingly, without any state change.
ii) If the operation is a write, I increment the ‘write’ statistic and initiate a switch case on the state of the cache. If the state is INVALID, I do a BusRdX snoop and then record a memory transfer or cache transfer based on the value of the parameter ‘copy’. For SHARED state, I do a BusUpgrade snoop and record a writeHit. For both the MODIFIED and EXCLUSIVE states, I only record writeHits. For each of the cases, I change the cache state to MODIFIED and increase execution times accordingly. I have also set in place appropriate error handling which helped me debug my program.
3) For the MESI_bus_snoop, I iterate over each processors cache (except for the processor which initiated the snoop) and check if the cacheLine for the address in question exists. If the line exists, I execute a switch case on the state of the line, and then increment relevant statistic values based on the type of the snoop. The logic of the switch case is as follows:
i) If the state is INVALID, I do nothing. This is the default case.
ii) If the state is SHARED, I flush on a BusRd, invalidate and flush on a BusRdX, and only invalidate on a BusUpgrade.

ii) If the state is MODIFIED, I flush and shift to a SHARED state on a BusRd, and for BusRdX I flush and invalidate the cacheLine.
iv) If the state is EXCLUSIVE, I flush and shift to a SHARED state on a BusRd, and for BusRdX I flush and invalidate the cacheLine.

b) MOESI
1) For the MOESI_processor_access() function, I first check if a cache line already exists for the address being accessed. In the scenario where the line doesn't exist, (this represents a cold-miss scenario) I create a cache line using the provided fillLine API and set its state to INVALID. This initial check serves the purpose of ensuring that a NULL value is not returned when I try to check the state of the cache line. I am not counting this invalidation towards the invalidation statistic as it is specific to my implementation.
2) Next, I first record the state of the address in the processor's cache. Based on this state & the r/w nature of the access, I make the following decisions:
i) If the operation is a read, I increment the 'read' statistic. If the state is INVALID, I initiate the BusRd snoop and increment the ReadMiss stat. Based on the function parameter value 'copy', I change the state to SHARED or EXCLUSIVE and increase the execution times accordingly. If the state is not INVALID, I record a readHit and increase the execution time accordingly, without any state change.
ii) If the operation is a write, I increment the 'write' statistic and initiate a switch case on the state of the cache. If the state is INVALID, I do a BusRdX snoop and then record a memory transfer or cache transfer based on the value of the parameter 'copy'. For both the SHARED and OWNER states, I do a BusRdUpgrade snoop and record a writeHit. For both the MODIFIED and EXCLUSIVE states, I only record writeHits. For each of the cases, I change the cache state to MODIFIED and increase execution times accordingly. I have also set in place appropriate error handling.
3) For the MOESI_bus_snoop, I iterate over each processor's cache (except for the processor which initiated the snoop) and check if the cacheLine for the address in question exists. If the line exists, I execute a switch case on the state of the line, and then increment relevant statistic values based on the type of the snoop. The logic of the switch case is as follows:
i) If the state is INVALID, I do nothing. This is the default case.
ii) If the state is SHARED, I invalidate the cache when a BusRdX or BusUpgrade is snooped.
ii) If the state is MODIFIED, I flush and shift to a OWNER state on a BusRd, and for BusRdX I flush and invalidate the cacheLine.
iv) If the state is EXCLUSIVE, I shift to a SHARED state on a BusRd, and for BusRdX I invalidate the cacheLine.
v) If the state is OWNER, I flush on a BusRd. For BusRdX snoops, I flush and invalidate the cache line, whereas for BusUpgrade snoops, I only invalidate the cache line.

Files modified:
1. Makefile (debugging traces)
2. Cache.h (modified snoop function arguments)
3. cache.cc (core logic)
4. main.cc (warnings fixed for clang version)