**Lab 2 Documentation**

**By Poulami Das, Yi-Shan Lu, Tian Zhang**

**# Work Distribution**

The design of the protocol was discussed together and Yi-Shan implemented the design source code.

The test cases were discussed and written together by Poulami and Tian.

**# MESI**

**Implement MESI protocol using c++**

1. A reasonable fix to get a higher hit rate would be to count the partial hits as full hits. This is a valid assumption because we are dealing with a single core.

2. The bus request created to store to a location that is invalid in cache is same as that to a location that is shared in cache. The reason behind doing so is simplicity. Store to a shared line is normally handled by invalidating other copies. However, reading the cache line again is functionally correct, and we can reuse the code for Store to an invalid line. If we follow the normal method, then there are more cases to handle, thus more difficult to verify the design.

To see why there are more cases, observe that the normal operation for storing to a shared line involves sending invalidation to the home site. The home site will now invalidate all the earlier shared copies and the processor writing to the cache line will have it in modified state. Store to an invalid cache line is trickier than this. The design needs to check if the same cache line is exclusive or shared or modified elsewhere, get the most recent correct copy and accordingly update the cache state for this line in every processor.

**Description of Protocol Implementation**

First of all, we only implement MODIFIED, SHARED and INVALID states for simplicity. Processors can only send to IUs READ requests for load/store misses; and WRITE requests for acknowledgements for invalidations and writebacks (if with permit\_tag MODIFIED).

A READ request is directed from the requester to its corresponding home site, or from the home site to the owner, i.e. the processor holding the cache line MODIFIED.

1. The home site sends out necessary INVALIDATIONs for READs with permit\_tag MODIFIED, or READs for READs with permit\_tag SHARED. Then the home site waits until all acknowledgements or writebacks come back, sets directories accordingly and replies back to the requester. Multiple messages for a single request are buffered and sent one by one in the following cycles, during which only replies can interleave.
2. The owner snoops its cache for the READ (with permit\_tag SHARED) to change the line to SHARED, and writes back to the home site using WRITE with permit\_tag MODIFIED.

An INVALIDATE request is always sent from the line’s home site for a READ with permit\_tag MODIFIED. The INVALIDATE is handled by snooping the cache to change the line to INVALID, and then acknowledging the home site using WRITE with the before-demotion permit\_tag in cache. If the WRITE is with MODIFIED, then it is also a writeback.

A WRITE request either results from lru\_replacement; or from responding INVALIDATE; or from handling READ to the owner. It is always sent to the home site. The WRITE is sent with the before-demotion permit\_tag in sender’s cache. The home site will maintain the directory, write back to memory accordingly, and send reply to the corresponding pending READ when appropriate.

A READ reply is always sent from the home site to its requester. It will satisfy the corresponding READ request and let the processor proceed.

A directory entry composes of a 32-bit bit vector and a dirty bit. The dirty bit indicates modification, and the bit vector indicates which processors hold the line. The directory structure is allocated dedicatedly and is smaller than 5 \* 768 = 3840 bytes, which is 120 4-word cache lines.

An IU handles messages with the following priorities:

1. Getting replies,
2. Sending messages left from a request,
3. Serving write request from the processor,
4. Retrying the existing net request (due to buffer space),
5. Serving new net requests,
6. Serving read request from the processor.

We add two queues per IU. reads\_to\_answer is for tracking reads to be answered; and msgs\_to\_send is for buffering multiple messages for a request. Both queues have 32 entries, respectively.

**Description of Test cases**

Our design has been verified using the test cases described next. We also have few extra test cases but that is very specific to our design and was used to check our design robustness when network latencies or other things change. These have not been included in the documentation or the final submission.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Test 0: Single Processor Read-Write Test**

The origin test case provided

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Test 1: Single Processor Read-Write Test**

This test is just to ensure the basic cache functionality.

Only one processor involved in this test

P1 x = load A;

Store 50 to A;

x = load A

Error case x != 50

Command: ./sim 1 10 0 1 | grep -q “fail”

If there is no fail in output, then it passes this case

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Test 2: Sequential consistency test of 2 processors**

This test aims to test sequential consistency

2 processors involved in this test cases

Initially A = B = 0

Processor 1: Store 200 to A, then store 1 to B

Processor 2: While(B == 0) ; load A

Error cases: A not equals to 200 in processor 2

Command: ./sim 2 100 0 1 | grep -q “fail”

If there is no fail in output, then it passes this case

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Test 3: Another sequential test of 3 processors**

This test aims to test sequential consistency

3 processors involved in this test cases

Initially, A = B = 0

P1: A = 1 P2: if(A==1) B = 1 P3 : if(B==1) load A

Error case A = 0 in processor 3

Command: ./sim 3 100 0 1 | grep -q “fail”

If there is no fail in output, then it passes this case

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Test 4: Bit vector check in the directory**

This test aims to test the complete bit vector implementation in the directory.

32 processors involved in this test cases

Every processor stores a value in range R to A, then every processor loads A.

Error case, if some processor load wrong value from A, which is not in range R

Command: ./sim 32 100 0 1 | grep -q “fail”

If there is no fail in output, then it passes this case

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Test 5: Expand test 2, sequential consistency for 32 processors**

This test aims to test sequential consistency for 32 processors

32 processors involved in this test cases

Initially everything is zero

Processor 0: A = 50; B = 1;

For other processors

Processor i A+i = 50+i ; B+i = 1; while(B+i-1 == 0) ; load A+i-1

Error cases: A+i-1 not equals to 50 + i - 1

Command: ./sim 32 1000 0 1 | grep -q “fail”

If there is no fail in output, then it passes this case

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Test 6: Expand test 3, sequential consistency for 32 processors**

This test aims to test sequential consistency for 32 processors

32 processors involved in this test cases

Initially, everything is zero

p1 : A = 1

p2 : if(A==1) B = 1

p3 : if(B==1) C = 1

…

p32 : if( Z == 1) load A

Error case A = 0 in processor 32

Command : ./sim 32 1000 0 1 | grep -q “fail”

If there is no fail in output, then it passes this case

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Test 7: Check directory similar to test 4**

This test aims to test address in directory protocol

32 processors involved in this test cases

Every processor stores to a random location, and every processor stores to different location.

Every processor now loads from that location and checks if the value is correct

Error case : some processor load the wrong value which is different to its stored value

Command : ./sim 32 1000 0 1 | grep -q “fail”

If there is no fail in output, then it passes this case

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Test 8: Store on a shared cache line followed by read on another**

This test is to test the functionality of directory and sequential consistency both.

Every one first load A, make a shared copy

P1: A = 50; B = 100

Other processor: While (A!=50); read B;

Error Case: B is not equal to 100

Command: ./sim 32 1000 0 1 | grep -q “fail”

If there is no fail in output, then it passes this case