ECE382N-10: Parallel Computer Architecture
Lab 2: Cache Coherency

Group Members
Sravya Satujoda (ss99384)
Sayam Sethi (ss225962)
Ishika Bhattacharya (ib5887)

Part 1

Why isn't the hit rate 0.5 or so? What would be a reasonable fix to get the hit rate, in this case, to 0.5?

In the given implementation, a cache load operation always requests the cache line in a Shared state (even if the cache line is invalid in all the other processors). If a subsequent store finds the cache line in the shared state, it is counted as a partial hit. Only in exclusive or modified states is it counted as a full hit. The hit rate only takes into consideration full hits.
We expect a hit rate of 0.5 because there are 512 unique addresses that can be accessed and 256 words (32 cachelines) that can fit in the cache. So every time an address is generated, there are 256 words in the cache and 256 not in the cache. The probability of the generated address being in the cache is 256/512 = 0.5 so the probability of it hitting is 0.5.
A reasonable fix for this would be for the processor to bring the cache line into the cache in Exclusive state on a load operation if no other processor has it in shared at that point.

Part 2

Note that the bus request created when you store to a location that is Invalid in the cache is exactly the same as the bus request created when you store to a location that is Shared in the cache. Why is this approach simpler? How is it normally done and why is it normally done that way?

If a cache line is in Shared state, it means that it already has the clean data available and all other processors have it in shared or invalid states. Normally, in this case we issue a Bus Upgrade or Get Permission that tells the other processors to invalidate that cache line in their caches if they have it in the shared state. If the cache line has it in the Invalid state, it needs to read the most up-to-date copy of the cache line from the other processors. The processor normally sends a Bus Read Exclusive or Read with Intent to Modify bus operation, and any cache having that line in the Modified state would need to flush the latest data onto the bus or update it in the home node, after which the data is sent back to the source processor. Having two separate bus operations saves us the unnecessary bus and memory transactions in the Get Permission case, since the clean data is already available. This is a real-world implementation that is generally used.

In the given implementation, in both Shared and Invalid states, a Read with Intent to Modify is sent to the home node. If a Get Permission was sent to the home node by a processor (P1) and another processor (P2) sends a RWITM request to the home node that reaches the home node

before the Get Permission request, then an Invalidate request is sent to P1. If this invalidate reaches P1 before the GP acknowledgement from the home node, P1 will invalidate the cache line. When P1 finally receives the GP acknowledgement and tries to perform the store to the cache line, it no longer has the cache line data.
If we were to have separate bus operations for both these cases, we would need to include support so that if an invalidate request is received before the acknowledgement for the get permission we sent out, the GP needs to be retried as a RWITM, which complicates the logic. Thus having the same busop which is RWITM for both the states of the cache line will make the implementation simpler in the Home node.

Part 3: Add Writeback (10%)
The current code has a writeback logic in cache->reply() for modified victim cache lines that are being replaced. Finish this write back implementation with correct write back address.

Cache->reply function is called to fill a new cache line into the cache. In this function, LRU replacement function gives the victim cache line to be evicted and if it is in modified state, it should be written back to the home node's memory. The address of the cache line to be evicted must be calculated correctly to write back the appropriate cache line.

This address can be calculated using the set and way given by the LRU replacement function. The address tag is acquired and shifted by address_tag_shift which is 3 as there are 8 sets in total and ORed with the set shifted by set_shift which is 3 as the cache line size is 8 words.

Thus the writeback address is given by
wb.addr = (tags[car.set][car.way].address_tag << address_tag_shift) | (car.set << set_shift);

Code Snippet:

```
void cache_t::reply(proc_cmd_t proc_cmd) {
  // fill cache.  Since processor retries until load/store completes, only need to fill cache.

  cache_access_response_t car = lru_replacement(proc_cmd.addr);

  if (tags[car.set][car.way].permit_tag == MODIFIED) { // need to writeback since replacing modified line
    proc_cmd_t wb;
    wb.busop = WRITEBACK;
    wb.addr = (tags[car.set][car.way].address_tag << address_tag_shift) | (car.set << set_shift);
    copy_cache_line(wb.data, tags[car.set][car.way].data);
    if (iu->from_proc_writeback(wb)) {
      ERROR("should not retry a from_proc_writeback since there should only be one outstanding request");
    }
  }

  NOTE_ARGS(("%d: replacing addr_tag %d into set %d, assoc %d", node, car.address_tag, car.set, car.way));

  car.permit_tag = proc_cmd.permit_tag;
  cache_fill(car, proc_cmd.data);
}
```

Part 4: Add Snoop (10%)
Implement snoop in cache_snoop.cpp.

Cache->snoop function is used by IU to communicate with cache. In other words, it facilitates the IU to snoop the cache.
Cache->reply is also used to communicate with cache but that is used to fill the cache with new cache lines or when the cache wants to write back in the case of LRU eviction.
While cache->reply deals with “promotion” of the permit_tags in the form of a reply to the proc_cmd, cache->snoop deals with the “demotion” of the permit_tags because of a message from the directory to ensure cache coherence.

Cache->snoop is used to handle other communications between the cache and IU. These communications include IU sending the invalidation requests and write-back requests to cache to modify the permit tag if it is in shared state or exclusive state or modify the permit tag and write back the data if it is in Modified state. We make use of the response that is returned by the cache_snoop function to determine the actions to be taken for different states of the cache lines.

Thus our cache->snoop function implements checking the state of the cache line, updating the permit tags on invalidation requests and writing back the data (or sending the acknowledgement) by calling from_proc_writeback functions in IU.

Code Snippet:

```
response_t cache_t::snoop(proc_cmd_t wb) {
  response_t response;

  cache_access_response_t car;
  if (!cache_access(wb.addr, INVALID, &car)) {
    response.hit_p = false;
  } else {
    response.hit_p = true;
  }

  switch (wb.busop) {
    case WRITEBACK: {
      if (!response.hit_p) { // only conclusion is that the cache line has been evicted and writeback is still pending (which is
        response.retry_p = false;
        return(response);
      }
    }
    copy_cache_line(wb.data, tags[car.set][car.way].data);
    case READ:
    case INVALIDATE: {
      if (response.hit_p) {
        modify_permit_tag(car, wb.permit_tag);
      }
      response.hit_p = true;
      if (iu->from_proc_writeback(wb)) {
        response.retry_p = true;
        return(response);
      } else {
        response.retry_p = false;
      }
      NOTE_ARGS(("%d: writeback for addr_tag %d with new permit_tag %d", node, car.address_tag, wb.permit_tag));
      return(response);
    }

    default:
      ERROR("invalid busop");
  }
}
```

Part 5: Directory Protocol (35%)
Design and implement a directory-based scheme. Maximize performance while minimizing directory state.

Choosing Priorities:
We define four kinds of messages that can travel on the network. Each kind of message travels on a different priority queue and hence it is very easy to distinguish between the different messages by the IU. The messages are:

1. **(PRI1) Cache Request:** This request is sent by the directory to the node asking it to "demote" its coherence status and send back an acknowledgement of demotion with the MODIFIED data (if any).
2. **(PRI2) Cache Reply:** This reply is sent by the node to the directory as a reply to a **Cache Request** message or on a writeback because of cache eviction. On receiving such a message, the directory simply updates its entry according to the provided message.
3. **(PRI3) Directory Request**: This message is sent by a node to the directory requesting a "promotion" in its coherence status.
4. **(PRI0) Directory Reply:** After the directory determines that the directory request can be processed without breaking coherence, that is, the other nodes have sent acknowledgements for demoting their coherence state, it sends a reply to the requestor node with the upgraded permissions. The permissions requested by the **Directory Request** are always honored (the requestor node will always get a positive reply from the directory at some point of time in the future).

Priorities are chosen in such a way that coherence updates are always prioritized over the directory updates. Additionally, coherence "promotions" are prioritized over "demotions" since it might be possible that the demotion request comes on the queue while the promotion is still being processed. If the demotion was at a higher priority, the node would attempt to process it even though the promotion was still being processed. However, the cache would not have the promoted permissions and hence it would be unable to acknowledge the demotion (since it would still be waiting for the promotion reply) which would lead to a hardware deadlock at the node. Also, to follow the assignment criteria and to ensure correctness, only one of the 4 messages are processed in any cycle. Even if there is a message available on the queue for a lower priority, it is not popped to avoid back-pressure on the node.

The IU first attempts to process any pending request from the processor (cache). Out of the two requests from the processor, IU prioritizes a writeback over a request for a cache line promotion. This is simply because processing a request for promotion might lead to another writeback which would lead to a hardware deadlock.

Directory Information:
The IU stores directory information using 40 bits (which fits within a single cache line and might require at most 2 cache line accesses if the entry begins towards the end of a cache line).

Therefore the total memory utilized for the directory information is 5120 bytes or 1280 ints, which requires 160 cache lines. The information stored in the directory is as follows:

1. **Modified bit:** this bit is set to 1 if the cache is present in MODIFIED (or EXCLUSIVE) state somewhere. Even though we require only one bit, to make addressing easier, we use the entire byte.
2. **Node mask:** this is a 32 bit number with the i^{th} bit being equal to 1 if the cache line is present in that node in a state other than the INVALID state.

Since the IU processes at most two requests (one from the processor side and one from the network side), it will update its directory entry state not more than twice (since the processor request might belong to its own directory). Therefore we will require at most 4 memory accesses per cycle for the directory updates and 2 memory accesses for reading the memory relevant to the cache line. Thus, the IU never accesses the memory more than 6 times in one cycle.

Implementation:

For processing a processor request, the IU first determines the destination node. If the destination node is itself, it processes the **Cache Reply** or attempts to store the request in the **Directory Request** variable. If there was a pending directory request, it will retry in the next cycle. If the destination node is a different node, it attempts to send the reply or the request to the network in the appropriate queue. If it fails, it retries in the next cycle.

For processing a network request, the IU first checks if there is a pending request for that priority already present. If yes, it attempts to complete the request. It might have to be retried for multiple reasons. For a **Directory Reply** or a **Cache Request** message, a retry would be needed if there is a pending proc writeback command since processing the reply or snoop might lead to another writeback command. The IU never has to retry a **Cache Reply** command since it just has to update its directory entry and we are always within the memory access limits. The IU would need to retry the **Directory Request** command until its directory entry is in the valid state to be able to give the *green flag* to the requestor node. The IU simply checks if the current directory state allows for the requestor node to proceed with its request (that is, data may be read in SHARED state iff it is not in MODIFIED elsewhere or it might be requested in EXCLUSIVE or MODIFIED state iff it is in INVALID state everywhere else).

Part 6: Test code (35%)

We have created 5 test cases and included them in our test.cpp and test.h files. The test case numbers range from 1 to 5. We also implemented the livelock checking in all of the test cases.

Test case 1:

Command:
./sim 32 10000 1 1

Description:
This test case involves all the processors and checks the network congestion. We exhaust the network fifo of the home node of a randomly generated address and see how the code is handling it.
For this to happen,

1) We are doing loads to the same randomly generated address in all the processors and waiting for all loads to complete.
2) Then we are doing stores to the same address in all the processors, each processor writing a different value to it.
3) Once all the stores are completed, we do loads in all the processors from the same address and now all the loads should return the same value.

This value will be the one that is stored by the last processor to write. This depends on the implementation. Ideal case is processor 0 will complete the last store as it is the last one to send its request to the home node according to the loop iteration in advance_time function. Since this varies depending on the implementation, our pass criteria would be all processors should read the same value. Otherwise it will fail.
Thus this checks the network congestion and the write backs.

Test case 2:

Command:
./sim <any number> 10000 2 1
This test case implements Dekker's algorithm for up to two processors and the critical section increments a shared memory location. Hence, effectively we count the number of processors (either 1 or 2). The check also verifies if the count variable is equal to the min(number of procs, 2). The memory locations are chosen such that they are in different cache lines and some of them also have different home nodes.

Test case 3:

Command:
./sim <any number greater than 1> 100000 3 1
This is a simple test to check if sequential consistency is ensured when loads in the opposite order happen to memory locations that have data stored to the memory locations. The loads on the processor begin after 10000 cycles so in expectation most stores should have

completed.Therefore, all values that are read by the other processor should be the new stored values.

Test case 4:

Command:

./sim <any number greater than 1> 100000 4 1

All processors execute loads and stores simultaneously to different memory locations. This also leads to a lot of messages on the network across all nodes and this is kind of a stress test for the implementation.

Test case 5:

Command:

./sim 8 1000 5 1

Description:

This test case checks if the LRU eviction is happening properly and if the directory updates in the home node are happening properly or not.

1) We issued one store and 3 consecutive loads in processor 7 to the addresses that point to the same set but with different address tags.
   Those addresses are 5, 65, 130, 195.
2) Simultaneously we are doing 4 loads to the same addresses in processor 5. So depending on the implementation, both the processors will have the cache line either in Shared state or Processor 7 will have it in Modified state and processor 5 will have it in Invalid state.
3) Then we do a load to address 260 which will force the cache line 0-7 to get evicted according to LRU and if it is in M state it will be written back to the home node's memory.
4) Then we do a store to the address 5 in processor 0 and once it is finished we do a load in processor 3 from the same address.

So now the processor 3 should see the value stored by processor 0.
Thus this testcase checks the LRU eviction, LRU eviction information communication to the home node and the write back implementation both in cache->reply and cache->snoop.