Benjamin Cho
Siavash Zangeneh
Behzad Boroujerdian
Lab 2 Report

Part a) The reason that the hit ratio is less than 50% is that in the current implementation, a load instruction brings in the cacheline in the SHARED state. Then even if the processors executes an instruction to the same cacheline, it would cause a “partial hit” not a “full hit”. The reason for the “partial hit” is that the IU still needs to invalidate all the other caches and therefore writing cannot be done without network access.

The solution for that is to bring in the data in cache in EXCLUSIVE state. A write to a cacheline with EXCLUSIVE state does not need communication with other nodes, and thus is a full hit.

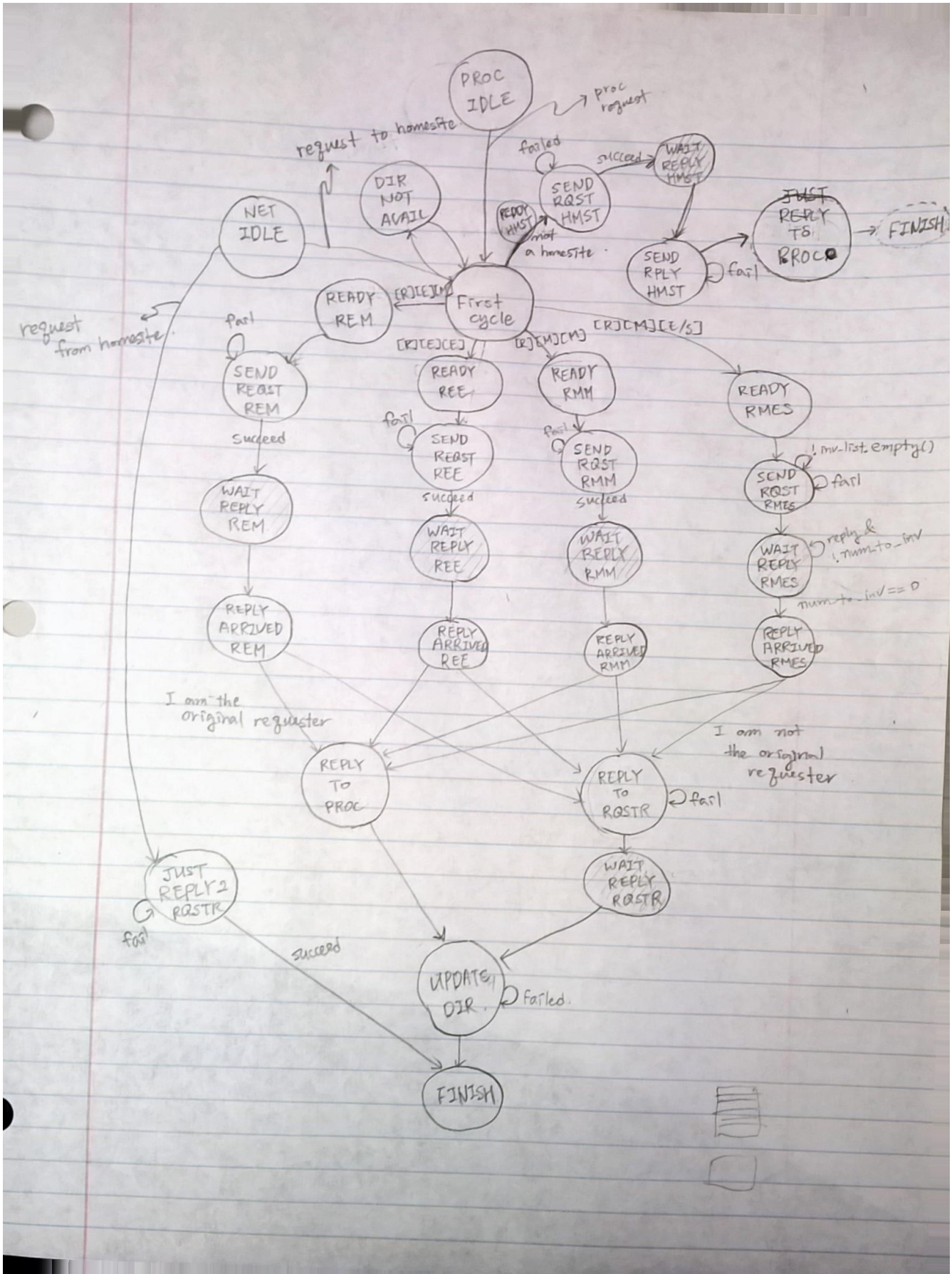
Part b) When writing to a cache in shared state, instead of asking for data from the memory, one can simply change the cache state to modified and invalidate all other caches. While this could work, it is hard to implement. For example, take the scenario of two processors writing to the same cacheline that is the SHARED state on both processors. Even if the system is designed to ensure that a cache is only modified after the acknowledgment is received from other processors that they're invalidated. If the second processor attempts to write to that cacheline before invalidation is received, then the IU should dequeue that SHARED-to_MODIFIED request from the processor and respond to it as if it a “read with intention to modify”. It is much easier to always request “read with intention to modify” regardless of the state.

Part c and d) The code is given in the sim/ directory. The way communication is handled is through some state machines in the IUs. A state machine for each ongoing access is allocated in the IU to rhandle network communication and the latency associated with that. The maximum number of state machines in each processor is set as 32, that allows each IU to handle the local cache request and any request from 31 other potential processors. For each cache miss or a “partial hit”, the directory needs to be consulted to know where to get the data / permission to modify the data. If the directory is not in the node that requested that data, a message is sent over the network to the homesite that contains the directory entry for that cacheline and the homesite deals with the request. All network communication are acknowledged to prevent any potential race conditions and sequential inconsistencies.
Note: we have introduced two new files: directory.h and behav_model.cpp. We could have merged both with our test.cpp and iu.h, however, we decided to keep them separated for the sake of cleanness of the code.

The state machine of the IU controller is attached at the end of the report.

Part e) The test cases are defined using arrays of structures that define the address and data of load/store instructions executed on different processors. We have two categories of test cases: some of them are completely deterministic and are evaluated by the data that are eventually loaded by the instructions. The other test cases depend on the order of executions. For them, we log the cycle number of the instructions that are executed and we feed the same instructions in the order they're executed to another model that simulates the cache lines, but does not simulate the network latencies. This behavioral model is used to verify the correctness of the network operations. Each test case contains comments in test.cpp that briefly explains what the test case is doing.
Note: behavioral model generates a logs.txt file where all the transitions for both the structural (the

cache designed by us or other teams) and the behavioral caches in terms of their tates are logged. As soon as a discrepancy is detected between the two, an error is generated and the program exits out.


PROC IDLE
proc request
request to homesite
DIR NOT AVAIL
failed
SEND RQST HMST
succeed
WAIT REPLY HMST
REPLY HMST
not a homesite
NET IDLE
SEND RPLY HMST
fail
JUST REPLY TO PROC
FINISH
request from homesite
READY REM
[R][E][M]
First cycle
[R][M][M]
[R][M][E/S]
[R][E][E]
fail
SEND REQST REM
READY REE
READY RMM
READY RMES
succeed
fail
SEND REQST REE
SEND REQST RMM
SEND REQST RMES
! mv_list.empty()
fail
succeed
succeed
WAIT REPLY REM
WAIT REPLY REE
WAIT REPLY RMM
WAIT REPLY RMES
reply & ! num_to_inv
num_to_inv == 0
REPLY ARRIVED REM
REPLY ARRIVED REE
REPLY ARRIVED RMM
REPLY ARRIVED RMES
I am the original requester
I am not the original requester
REPLY TO PROC
REPLY TO RQSTR
fail
JUST REPLY2 RQSTR
WAIT REPLY RQSTR
fail
succeed
UPDATE DIR
failed
FINISH