15-418 Final Project Report

Emily Szabo Sophia Qingyang Cao| Apr 28, 2025

Summary

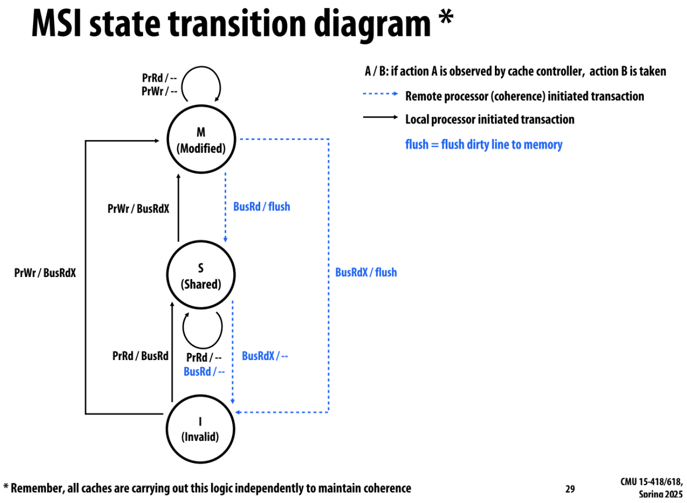
We implemented MSI, MESI, MOESI, and MESIF snooping-based cache coherence protocols on the Shark machines. To better understand the scalability and optimizations each protocol makes on different workloads, we also implemented monitoring to gather additional statistics about cache misses, interconnect traffic, invalidations, and flushes to main memory.

Background

In uniprocessor systems, caches can simply load/store and evict data without having to notify other parts of the system about their actions; however, when you move into parallel systems, this becomes a lot more challenging because all of the processors in the system have to communicate with each other regarding what data they're writing into or reading from to maintain memory consistency across all of their caches.

Different protocols have been implemented for caches across multiprocessor systems to maintain memory consistency, including snooping-based and directory-based protocols. Our project focuses on implementing several snooping-based protocols to gather stats about a multiprocessor system's caches. Snooping-based protocols maintain memory consistency by storing metadata about each cache line: each line can be in a particular state, and depending on networking traffic from the cache controller on the local processor or the interconnect between caches that lets the local processor listen to activity from remote machines, cache lines can transition states.

The four protocols we implemented are MSI, MESI, MOESI, and MESIF. Each of these protocols exists as a development on a previous snooping-based protocol to optimize a particular memory access pattern more effectively than the previous protocol. MSI is built on the MI protocol, and it adds a Shared “S” state to the existing Modified “M” and Invalid “I” states to allow sharing lines between different processors, reducing invalidations and memory fetches. The following state diagram, credit to the 15-418 cache coherence lecture, describes the MSI protocol.



MSI state transition diagram *

A / B: if action A is observed by cache controller, action B is taken

Remote processor (coherence) initiated transaction

Local processor initiated transaction

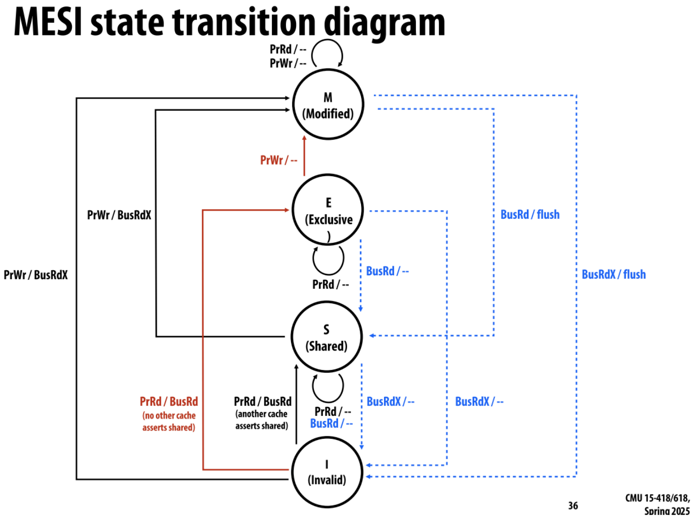
flush = flush dirty line to memory

* Remember, all caches are carrying out this logic independently to maintain coherence

29

CMU 15-418/618, Spring 2025

MESI builds upon the MSI protocol by adding an Exclusive “E” state that allows a processor that is the only one to be reading from a given cache line the ability to silently upgrade to the Modified “M” state, reducing interconnect traffic for workloads that have read-write sequences. The following state diagram, again credit to the 15-418 cache coherence lecture, describes the MESI protocol.

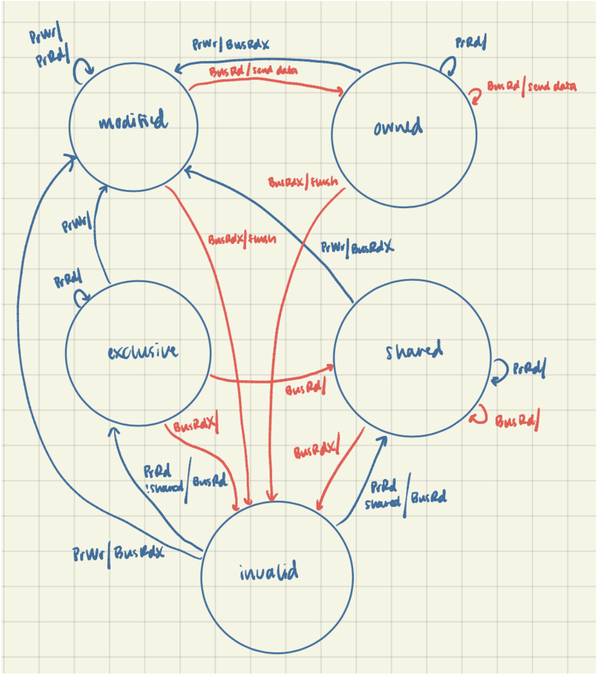


MESI state transition diagram

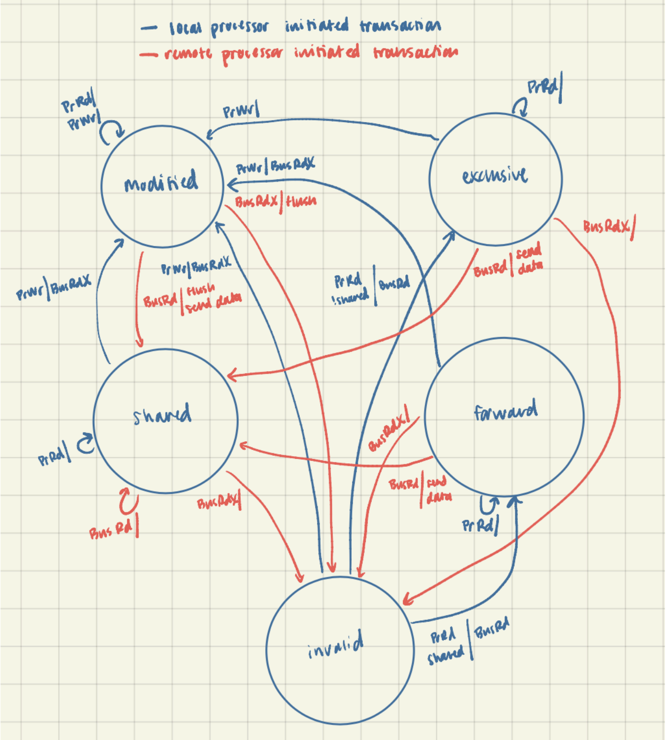
36

CMU 15-418/618,
Spring 2025

MOESI and MESIF both build on the MESI protocol to reduce memory fetches via caches supplying the data themselves, allowing for more optimal performance in workloads with a significant amount of memory sharing between processors. MOESI introduces an Owner "O" state that is transitioned to after being in the Modified "M" state as a designated processor to service read misses instead of immediately flushing the dirty data to memory. The following diagram was produced by our team, illustrating MOESI's state transitions.



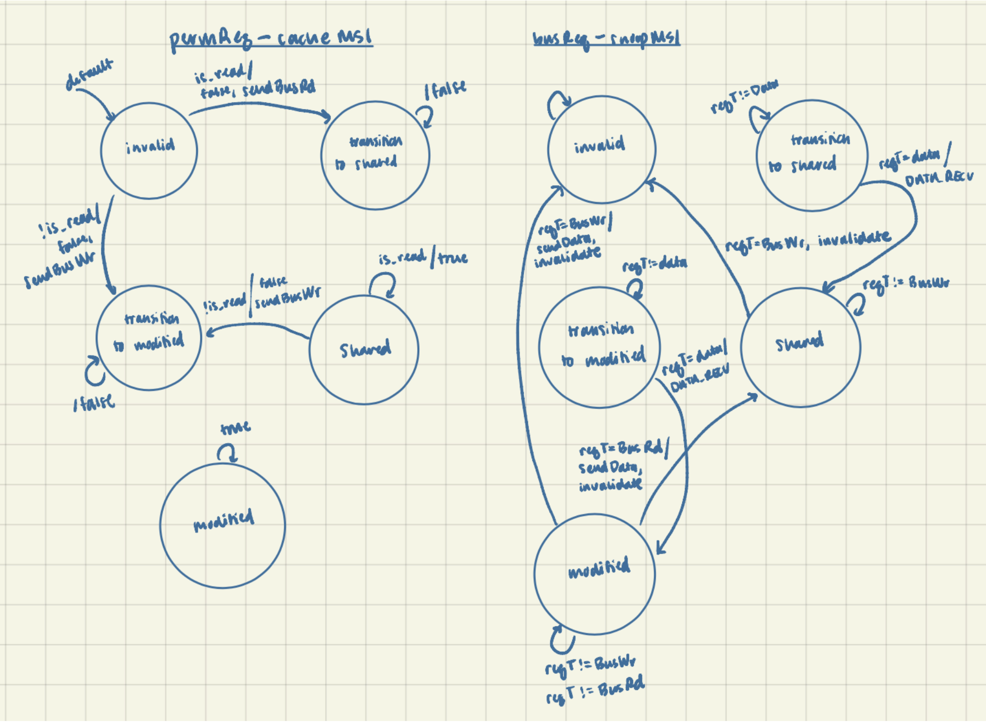
MESIF introduces a Forward “F” state that the last processor to have a read miss on a particular line enters, where it becomes designated to serve the next future read miss (and then the processor that has the next read miss becomes designated to serve the following read miss, and so on). The following diagram, also produced by our team, demonstrates the MESIF state transitions.



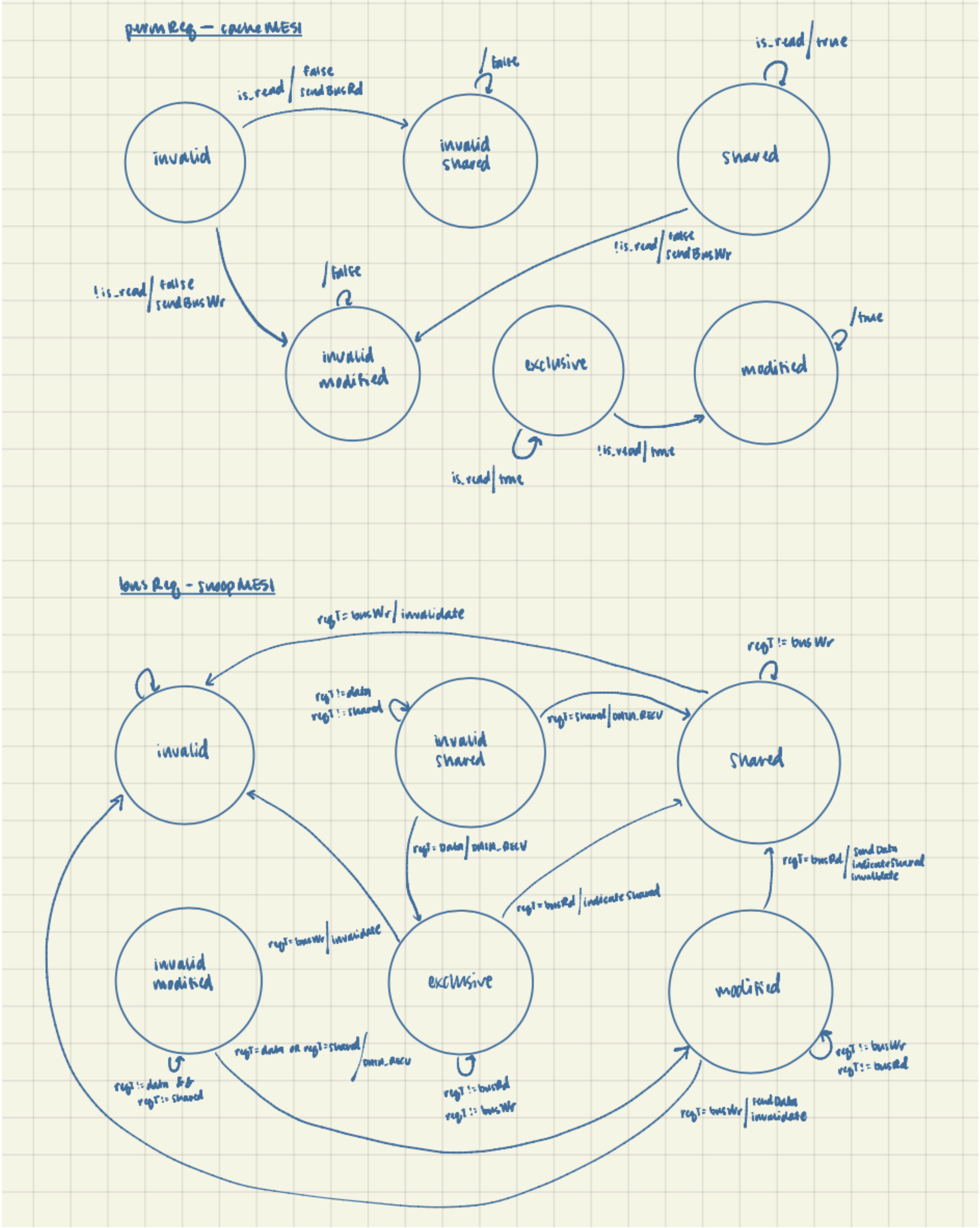
Approach

We built our project on the existing CADSS starter code in C for 15-346 P5. The existing code provides a working MI implementation. Because state transitions are not atomic when looking at implementing these snooping-based cache coherency protocols, a major difference between the theoretical state transition diagrams depicted in the previous section and the actual implementation diagrams is that there needs to be intermediate states to account for this lack of atomicity. For example, in the MI starter code, there is an INVALID_MODIFIED transition state that lines enter when they want to transition from Invalid to Modified but have not yet received the data from memory. To correctly implement MSI, MESI, MOESI, and MESIF in C in this codebase, we introduced two transition states: INVALID_SHARED and INVALID_MODIFIED. INVALID_SHARED serves as the transition state from Invalid to Shared, and later also optionally to Exclusive once implementation goes past MSI. INVALID_MODIFIED serves the same purpose as it does in the simple MI implementation.

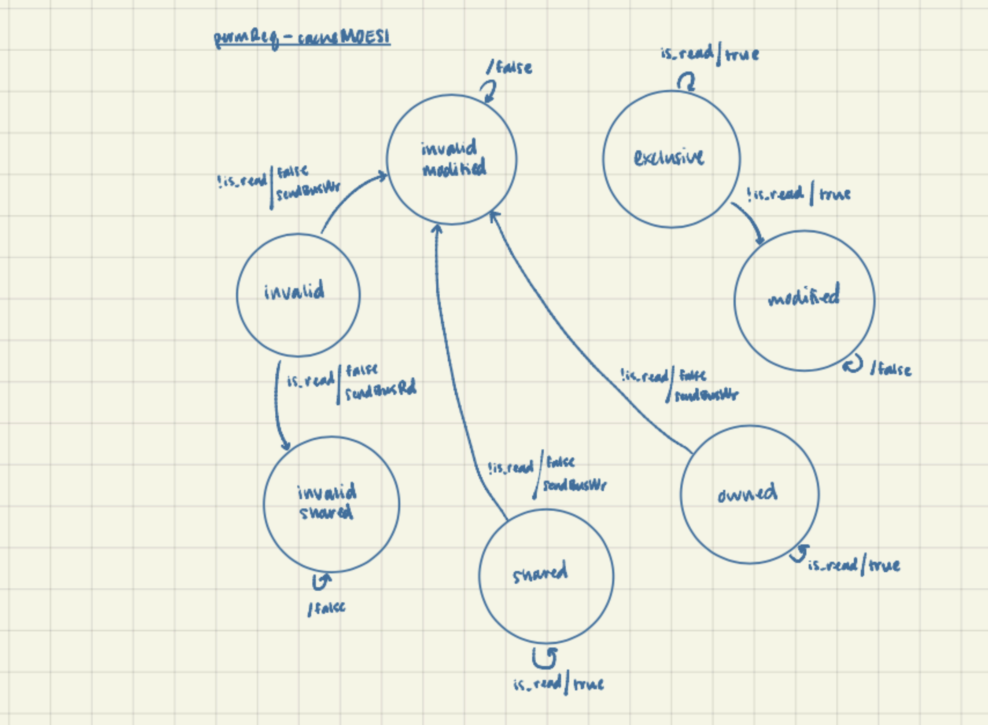
We put a lot of work into drawing out these state transition diagrams for implementation, and we split the diagrams into two diagrams instead of one per protocol as the existing codebase is developed in two pieces: (1) listening to requests from the local cache controller and (2) listening to activity on the interconnect to hear remote processors. Listening to local requests is denoted by cacheMX() functions, and snooping remote processors' activity is denoted by snoopMX() functions, where X serves as a placeholder for the particular snooping protocol. Here is our modified diagram for MSI implementation:

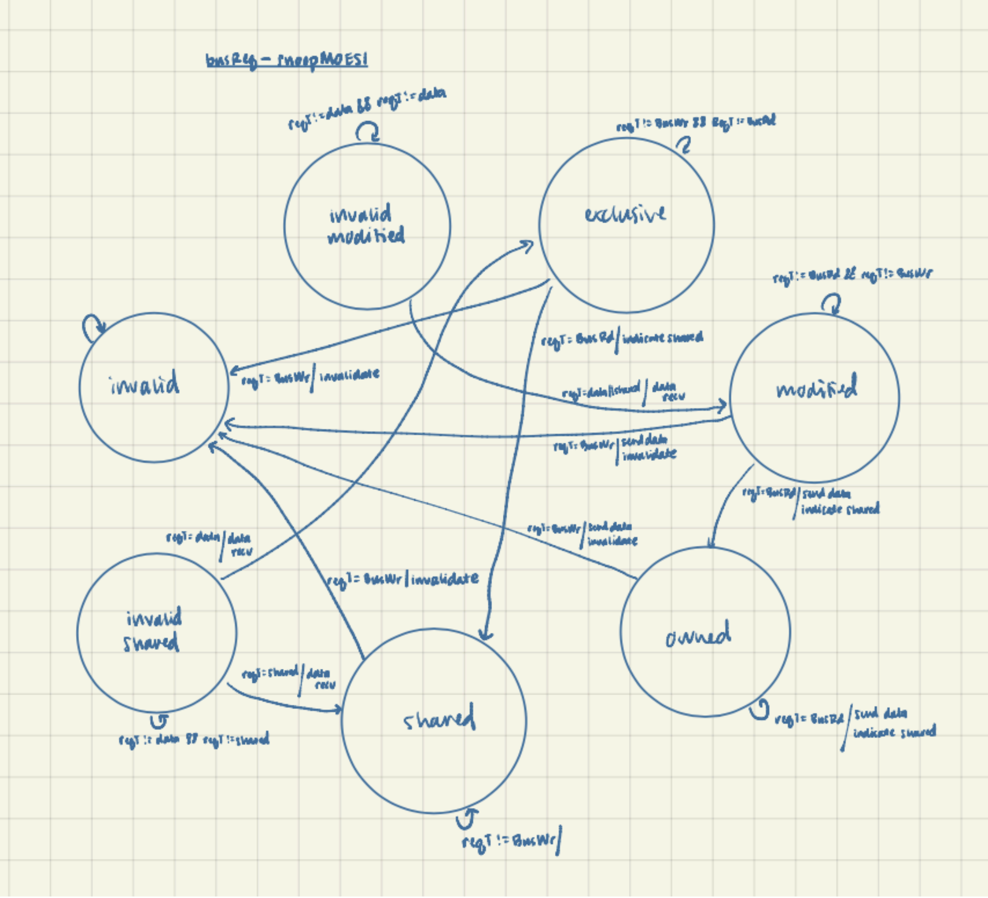


Here is our modified diagram for MESI implementation:

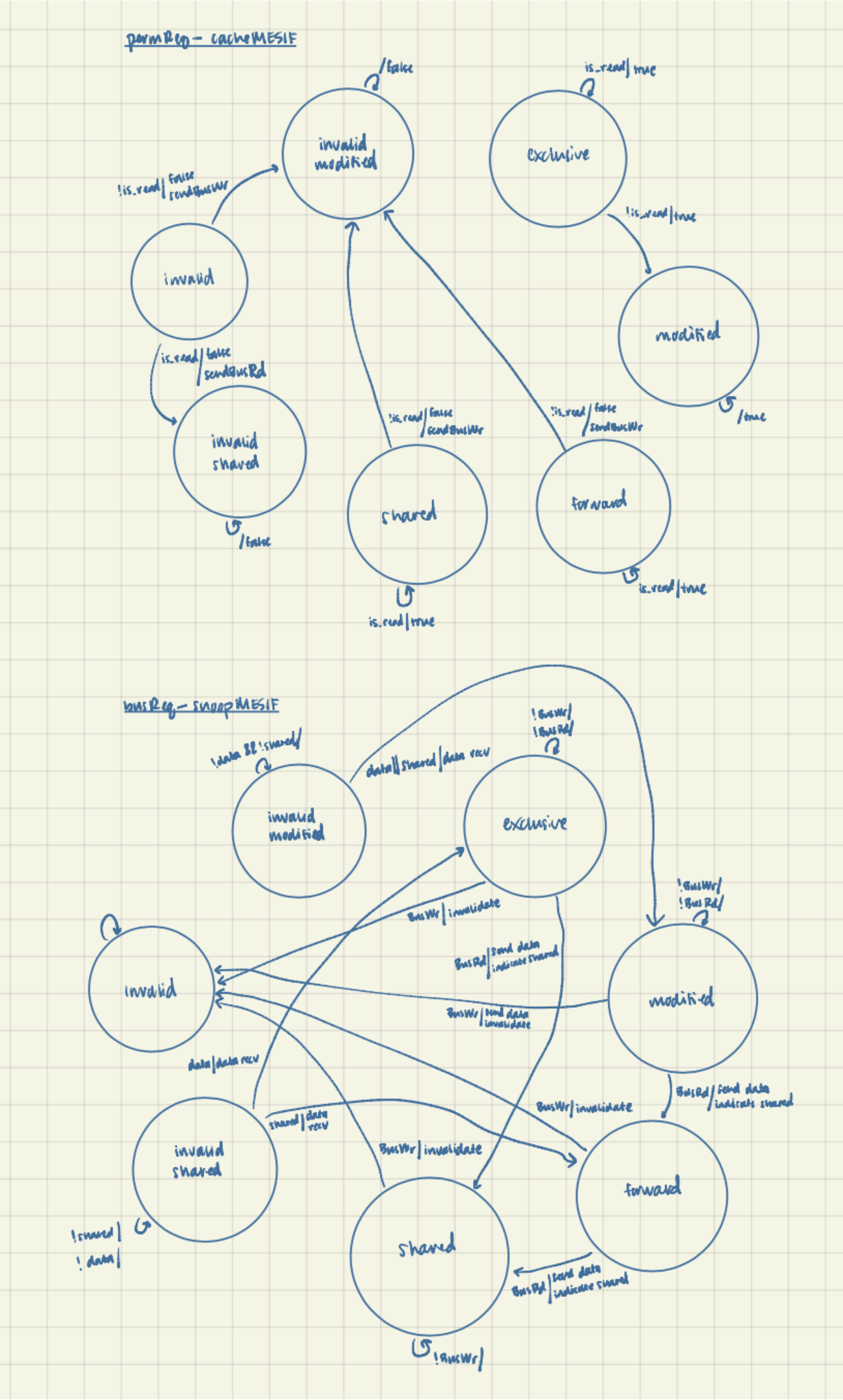


Here is our modified diagram for MOESI implementation:




busReq – snoopMOESI
reqT != data && reqT != data
reqT != BusWr && ReqT != BusRd
invalid modified
exclusive
reqT != BusRd && reqT != BusWr
reqT = BusRd | indicate shared
invalid
reqT = BusWr | invalidate
reqT = data||shared | data recv
modified
reqT = BusWr | send data invalidate
reqT = BusRd | send data indicate shared
reqT = BusWr | send data invalidate
reqT = data | data recv
reqT = BusWr | invalidate
invalid shared
owned
reqT = shared | data recv
reqT != data && reqT != shared
shared
reqT = BusRd | send data indicate shared
reqT != BusWr |

Here is our modified diagram for MESIF implementation:



After being able to correctly ideate on the lack of atomicity in actual hardware, we were able to successfully implement all four snooping-based protocols and gather results discussed in the next section.

Results

We evaluate the five implemented cache coherence protocols (MI, MSI, MESI, MOESI, and MESIF) across two workloads: 4proc_migratory and 4proc_prodcons. We compare protocols' performance based on several key metrics: total ticks, cache misses, bus transactions, invalidations, and flushes to memory.

Metrics Definitions

Before we jump into the result analysis, we explain our definition of some metrics.

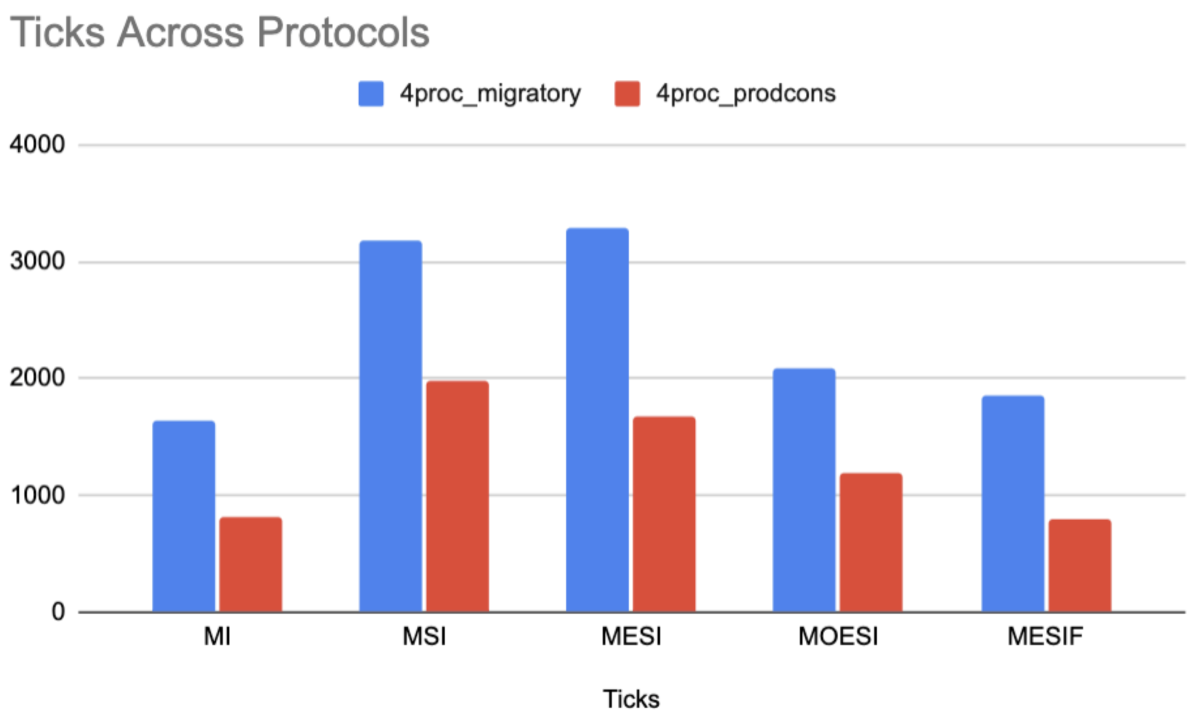
- Cache misses: if the action of write/read cannot be immediately taken
- Bus transaction: Requests to read/write passed on the bus
- Invalidations: wherever we need to invalidate a cache line, i.e. *ca = INVALIDATE
- Flush: **different from invalidation. Only when the dirty cache is invalidated, we need to flush to memory. E.g. invalidating a clean SHARED_STATE line does not need a flush.

Traces Features

4proc_migratory: Migratory sharing; Frequent transfers of write ownership across processors. Same memory address is repeatedly read/written by different processors.

4proc_prodcons: Producer-consumer; One processor writes to memory locations, followed by multiple processors reading the data without modifying it. Little to no ownership movement.

Ticks Across Protocols



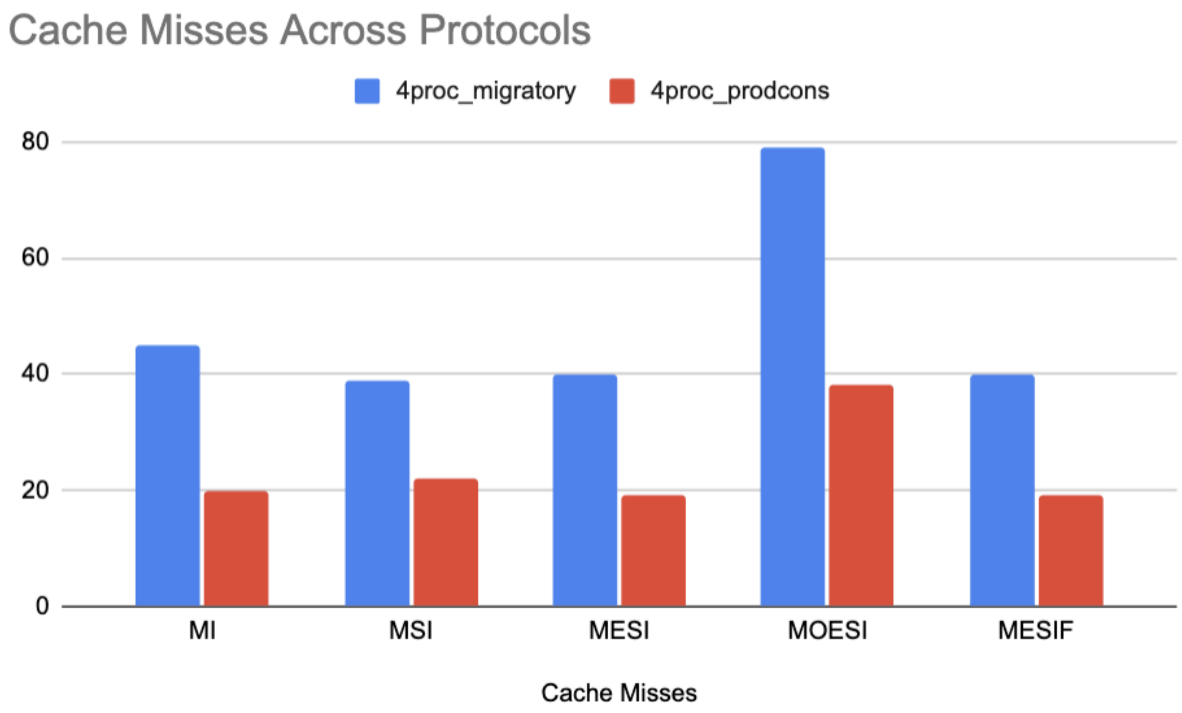
Observation

MI, MOESI, and MESIF show significantly fewer ticks than MSI and MESI, particularly in 4proc_migratory. In 4proc_prodcons, MESIF achieves the lowest tick count, while MI also performs relatively well.

Analysis

MI allows memory accesses with minimal logic and cordination, resulting in fewer ticks. MSI and MESI enforce invalidations on every write, introducing stalls and leading to higher tick counts, especially in migratory workloads where ownership changes frequently. MOESI and MESIF optimize ownership transitions: MOESI enables dirty sharing through the Owned state, reducing memory writebacks. MESIF designates a Forwarder for read-sharing, minimizing bus contention. In migratory workloads, MESIF achieves better performance by efficiently supporting frequent ownership changes without triggering full invalidations or memory flushes.

Cache Misses Across Protocols



Cache misses indicate how often cores needed to access the bus/memory to fetch data.

Observation

4proc_migratory:

MI, MSI, MESI, MESIF have ~40 misses. MOESI has significantly more (~80 misses).

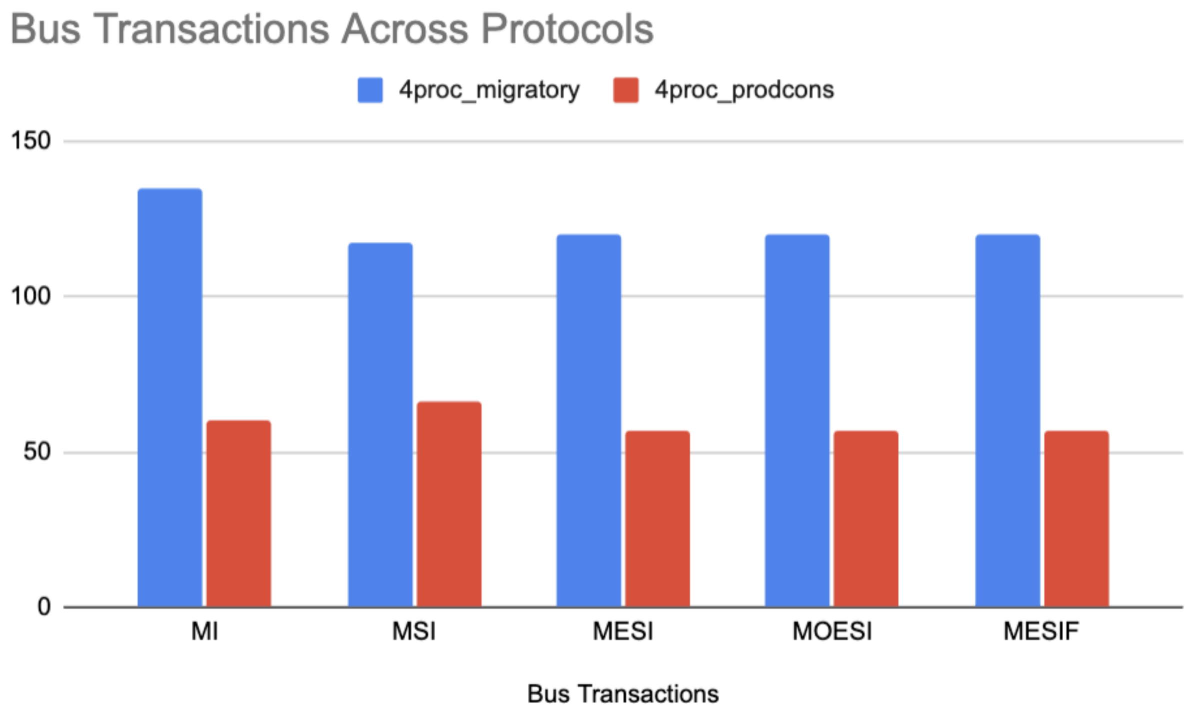
4proc_prodcons:

All protocols have relatively few misses (20–40). MESIF and MESI have the fewest misses (~19).

Analysis

Cache misses primarily reflect spatial and temporal locality rather than coherence mechanisms. MOESI's higher miss rate in migratory workloads suggests possible suboptimal ownership handoff or an implementation issue where cache lines are downgraded unnecessarily. MESIF and MESI maintain lower miss rates because Exclusive and Forward states reduce unnecessary invalidations and promote efficient data sharing. 4proc_prodcons naturally has fewer misses due to the producer-consumer pattern – each consumer accesses data that remains cached after producer writes.

Bus Transactions Across Protocols



Observation

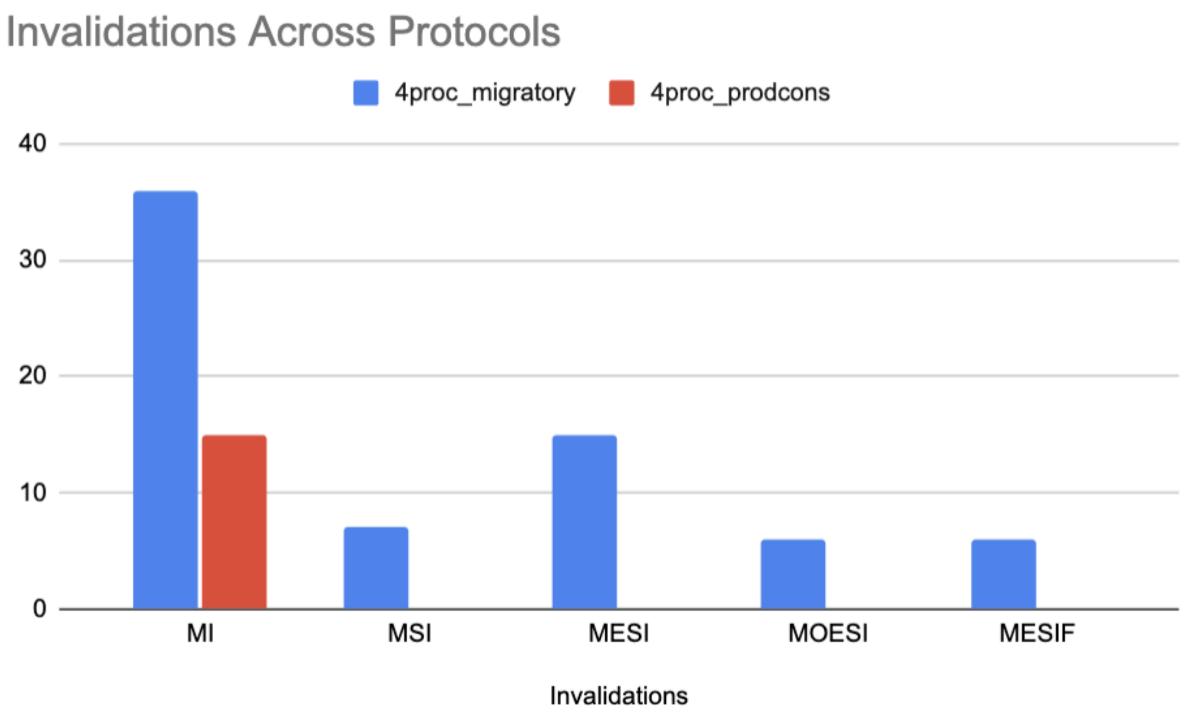
MI generates the most bus transactions across both traces. MESIF and MOESI consistently show slightly lower bus traffic compared to MI, MSI, and MESI. Overall, the bus transactions do not vary significantly across protocols.

Analysis

MI issues more frequent bus transactions because it does not optimize ownership states; every read or write tends to trigger visible traffic. MSI introduces invalidations on writes, slightly reducing bus activity compared to MI under trace 4proc_migratory. MESI intends to improve by reducing requests when data is in the Exclusive state. MOESI and MESIF are most efficient:

MOESI supports cache-to-cache data sharing without immediate memory involvement. MESIF's Forward state ensures that only one cache responds to BusRd, avoiding redundant data transmissions. Thus, MESIF and MOESI minimize unnecessary bus transactions, improving scalability, though not very obvious under the two traces.

Invalidations Across Protocols



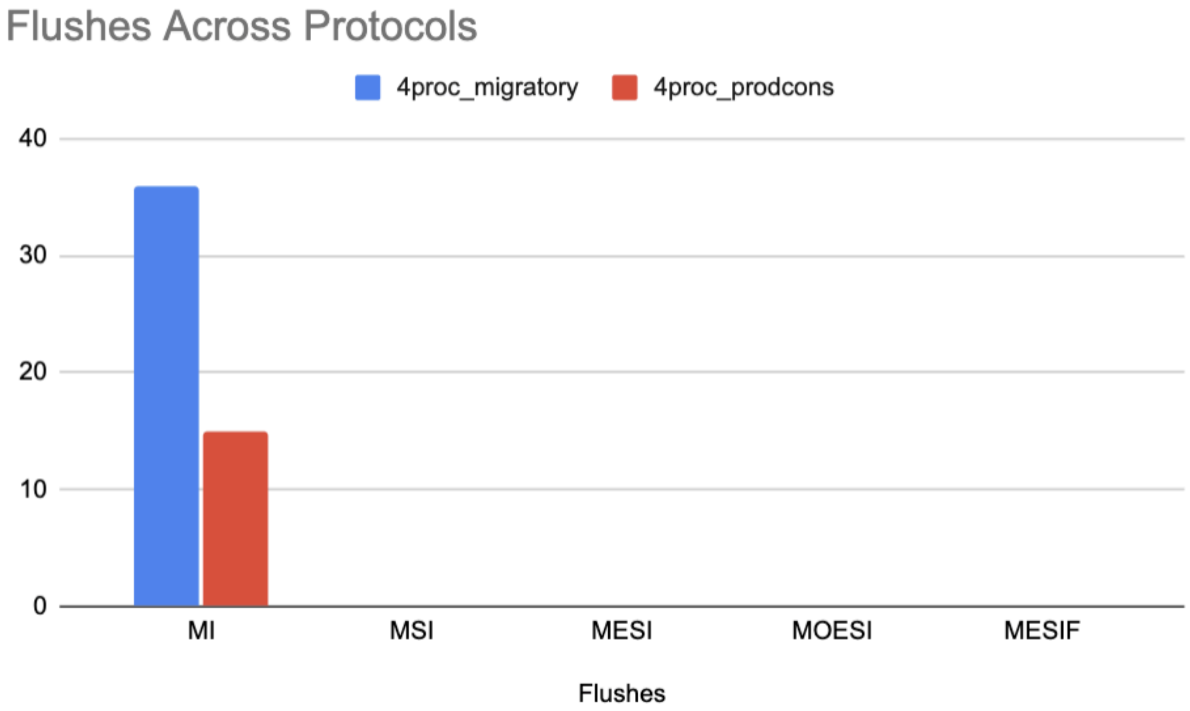
Observations

MI exhibits the highest number of invalidations in both traces. MSI, MESI, MOESI, and MESIF drastically reduce invalidations.

Analysis

MI’s simpler design treats many operations as requiring invalidation. MSI, MESI, MOESI, MESIF track data sharing more carefully: MSI invalidates on every write but avoids needless invalidations on reads. MESI further optimizes by using Exclusive state to defer invalidations. MOESI and MESIF minimize invalidations even more by permitting multiple caches to retain clean or owned copies where appropriate. In producer-consumer patterns, where data is mostly read after production, protocols like MESI and MESIF are particularly advantageous by avoiding invalidations altogether after initial writes.

Flushes Across Protocols



Observation

Only MI recorded significant flushes. MSI, MESI, MOESI, and MESIF show zero flushes across both traces.

Analysis

MI, lacking fine-grained dirty state management, must conservatively flush modified data back to memory frequently to maintain coherence. In contrast, MSI, MESI, MOESI, and MESIF track dirty lines through the Modified or Owned states: Only when a dirty line is invalidated or evicted is a flush necessary. In these traces, careful ownership tracking avoids unnecessary flushes altogether. The complete absence of flushes in MSI and beyond demonstrates the effectiveness of explicit dirty-state tracking in more advanced protocols.

References

[1] CADSS Codebase - https://github.com/bprail/cadss_public

[2] 15-418 Lecture 12: Snooping-Based Cache Coherency - https://www.cs.cmu.edu/afs/cs/academic/class/15418-s25/public/lectures/12_cachecoherence1.pdf

[3] 15-418 Lecture 14: Snooping Implementation - https://www.cs.cmu.edu/afs/cs/academic/class/15418-s25/public/lectures/14_snoopimpl.pdf

Work List + Distribution

Emily - Project Proposal, MSI theoretical + implementation diagrams, MESI theoretical + implementation diagrams, MOESI theoretical + implementation diagrams, MESIF theoretical + implementation diagrams, MESI code implementation, MOESI code implementation, MESIF code implementation, metrics graphing

Sophia - MSI code implementation, MSI code testing, MESI code testing, MOESI code testing, MESIF code testing, metrics code implementation, metrics gathering, metrics analysis, poster structure

Together - Project Milestone Report, Final Project Report

Intended Credit Distribution - 50/50