Cache Coherence Enforcement Protocol Simulation & Evaluation

Siddhant Kulkarni*, Ritesh Sangurmath#, Ranjan Yadav+
Department of Computer Science and Engineering
University of Colorado at Denver
Denver, Colorado
{* siddhant.kulkarni, # riteshprabhakar.sangurmath, + ranjankumar.yadav}@ucdenver.edu

I. INTRODUCTION

Since the introduction of multiprocessor architectures, cache coherence has been one of the major concerns focused on by many of the researchers [1, 12, 13]. This issue is illustrated in figure 1. Fact is that coherence issues are a fact of life. There will always be inconsistent states among data being shared by dedicated caches of different processors. In order to maintain consistent access to data for all processors, researchers have proposed several different algorithms or approaches. These approaches are mainly categorized into three different categories as shown in figure 2.

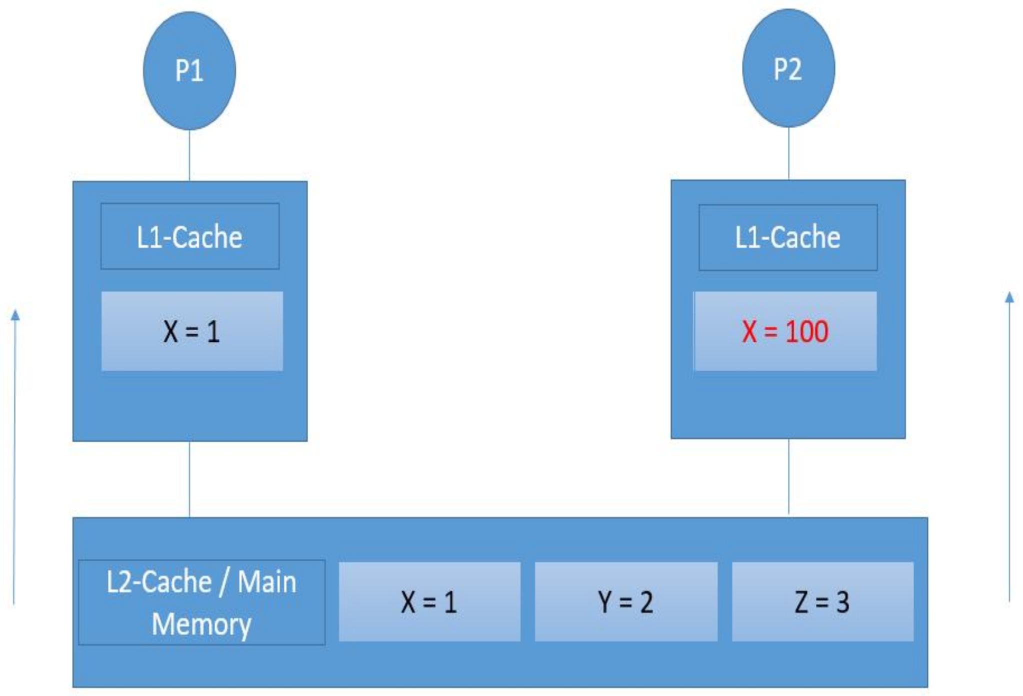


Figure 1. Issue of Cache Coherence

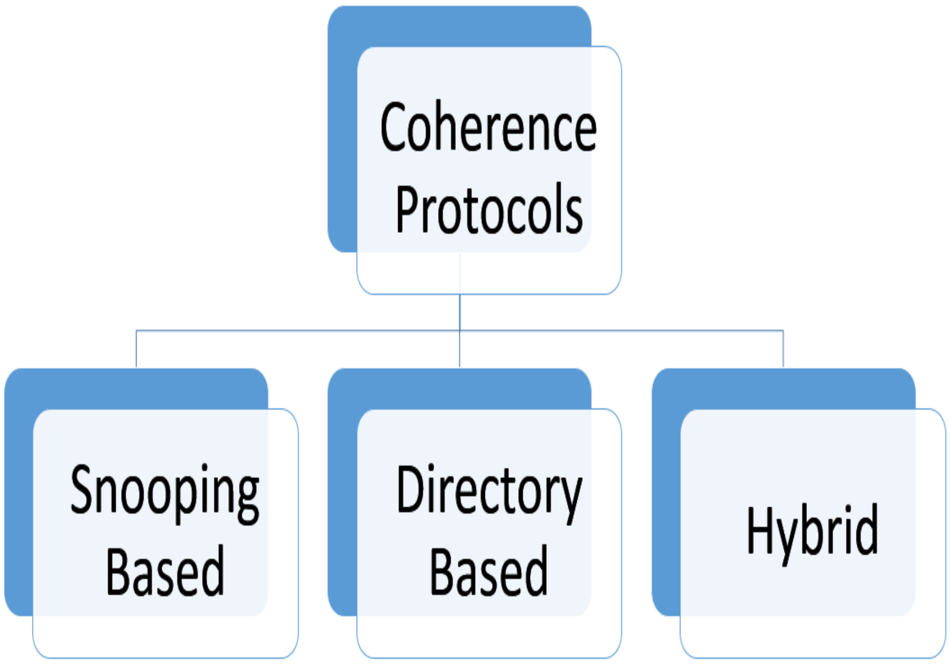


Figure 2. Classification of cache coherence enforcement approaches

Snooping based protocols are focused majorly on the use of buses to communicate among the different processors that are involved in the multiprocessor architecture. In this approach, the processors inform each other regarding the operation which they are going to attempt to carry out. Every processor maintains the state of each block in its local cache. This architecture is certainly limited in terms of scalability as it is highly dependent on the bus bandwidth. In order to overcome the issue of scalability, researchers turned towards a bit more centralized approach in the form of a directory based coherence enforcement technique. In this approach, a centralized directory maintains the state of each block along with the location of that block in local caches of the processors. Although this solves the concern of scalability, it is significantly slower that the snooping based protocols.

Given these constraints enforced by the snooping and directory based protocols, different researchers have begun exploring the idea of a hybrid solution. One of the most interesting ways in which a hybrid approach has been to use snooping based protocols when sufficient bandwidth is available and to use a directory based approach otherwise[15].

As a part of this project, we have focused on simulating snooping based protocols and thus rest of this report will focus on the same. In the next section we elaborate the 6 snooping based protocols that we have implemented and evaluated as a part of this project.

II. Snooping Based Protocols

We will elaborate snooping based protocols in two categories as follows: Write Update and Write Invalidate.

1. <u>Write Update</u>

The protocols that come under the Write-Invalidate consists of an invalid state. Write-invalidate will occur when a modification is done on the block which is shared between two processors. For instance, when a particular processor p1 make a modification (local write) on the block which is shared with another processor p2, then the processor p1 will send an invalid message to processor p2 which make the processor p2 block go to an invalid state.

There are four protocols that comes under Write-Invalidate:

- ➔ MSI (Modified, Shared, Invalid)
- ➔ MOSI (Modified, Owner, Shared, Invalid)
- ➔ MESI (Modified, Exclusive, Shared, Invalid)
- ➔ MERSI (Modified, Exclusive, Read, Shared, Invalid)

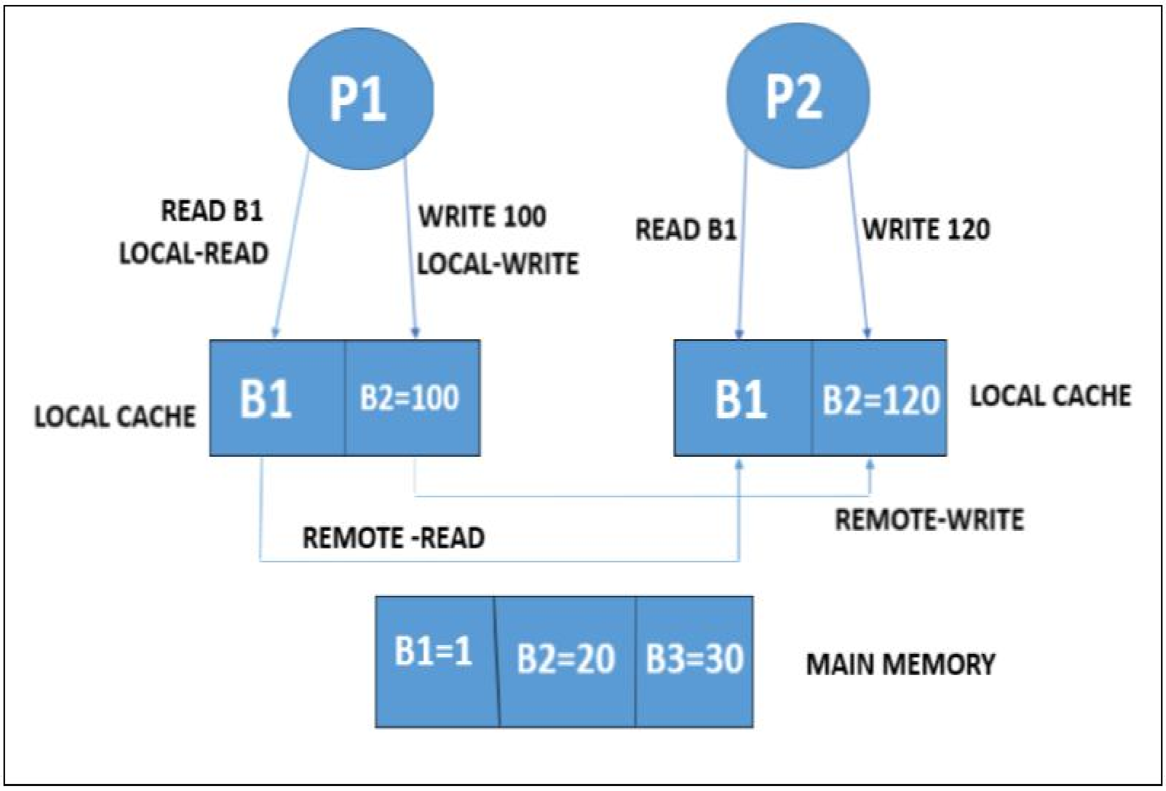


Figure 3. Functionality of snooping based protocols

Operations:

- **LR (Local Read) -** This operation is performed when the processor wants to read the block present in its own local cache. If we consider the above diagram processor p1 reads the block B1 which is present in its own local cache.
- **LW (Local Write) -** This operation is performed when the processor wants to write or modify the block present in its own cache. As per the above diagram processor P1 writes the value 100 to block B2 which is present in its own cache.
- **RR (Remote Read) -** This operation is performed when the other processor wants to read the block present in another processor. As per the above diagram processor P2 reads the block B1 which is present in the local cache of processor p1.
- **RW (Remote Write) -** This operation is done when the other processor wants to modify the block which is present in another processor. According to the diagram given above processor p2 will modify the block B2 which present in the local cache of processor p1.

a. MSI

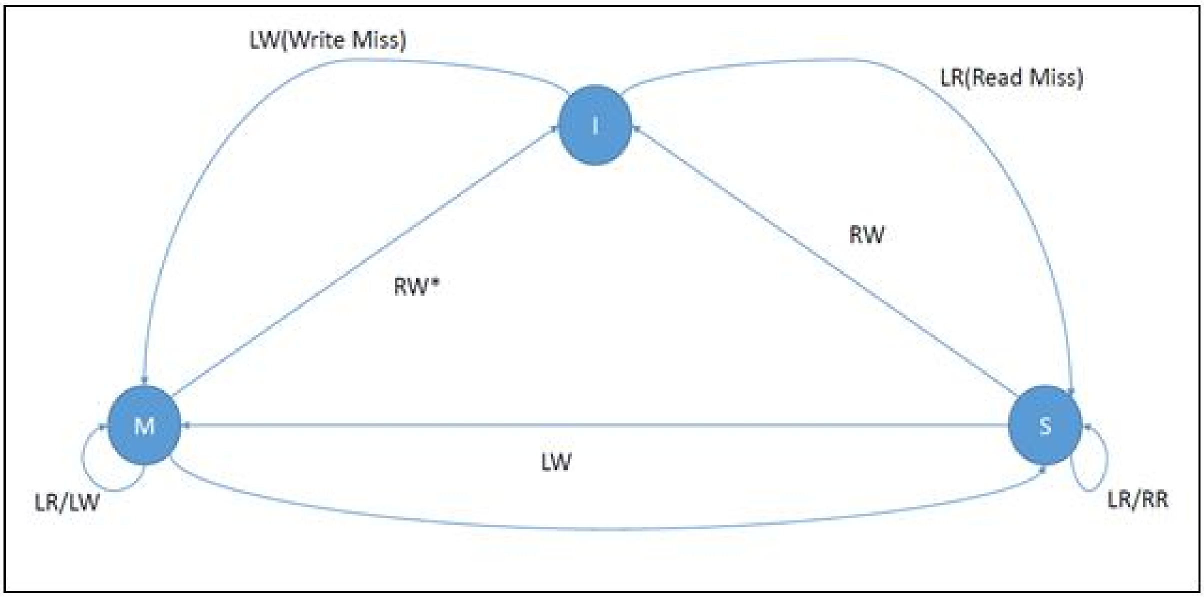


Figure 4. State Transition Diagram for MSI Protocols

MSI works in multiprocessor architecture. AS the name clearly indicates MSI has three state.

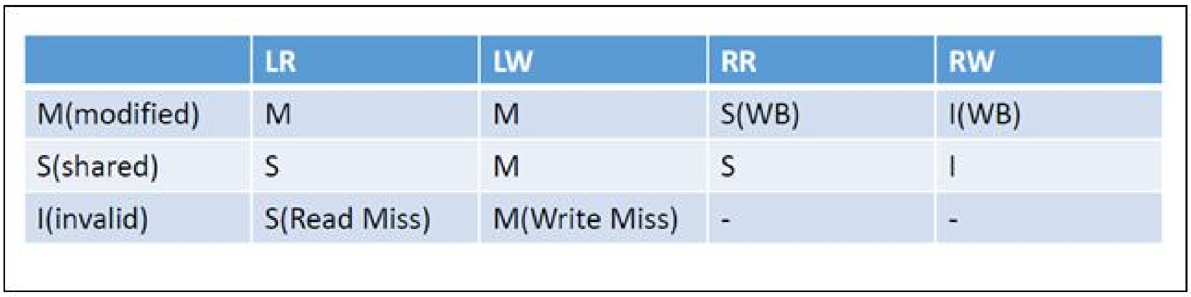
- **Modified**: When the processor writes new value to the block then the block will be in the modified state. And it will be the responsibility of the processor to write back the modified value to block present in the main memory to ensure cache coherence.
- **Shared**: The block will be in shared state when the block has been shared between two processors.
- **Invalid**: The block in an invalid state indicates that this block is not valid. The processor has to either get the updated/valid block from main memory or from processor which has the modified/updated block.

Transition Diagram:

The state transition diagram in figure 4 shows the state transition of the block. As the protocol name indicates there are three states. If we perform the operations such as Local-read and local-write on the block which is in modify state, then the block will be in the modify state, where as if we perform operations such as remote-read the block will be moved to shared state because the block will be shared between two processors and the operation like remote-write on the block in modify state will make the block to make the transition to invalid state because the processor will be having invalid data in the block.

The block in the invalid state causes a coherence cache miss. If the operation like local-read is performed on the block which is in invalid state causes read-miss and after fetching the block from main memory or from another processor the block will be moved to shared state and the operation like a Local-write performed on an invalid block causes write miss and after fetching the data from the main memory or from another processor the block will be moved to the modify state.

The block in shared state indicates the block is shared between two processors. The operations like local-read and remote read on the block in shared state makes the block to stay in the shared state, whereas the operation like remote-write makes the block to move to invalid state because the other processor has modified the block and the block which the processor is having is invalid and the operation like local-write on the block makes the block to move to modify state.



	LR	LW	RR	RW
M(modified)	M	M	S(WB)	I(WB)
S(shared)	S	M	S	I
I(invalid)	S(Read Miss)	M(Write Miss)	-	-

Figure 4. State Transition Table for MSI Protocols

b. MESI

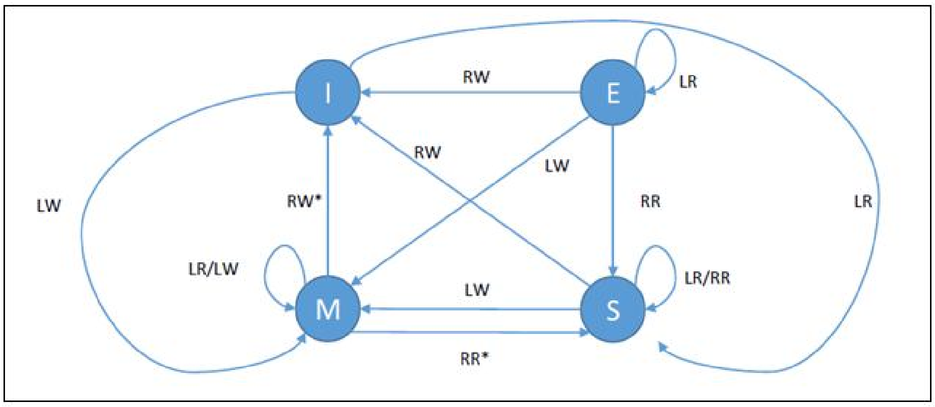


Figure 5. State Transition Diagram for MESI Protocols

MESI Protocol is also an extended version of MSI but it consists of an extra state called exclusive state. MESI protocol consists of four state:

- **Modified**: When the processor writes new value to the block then the block will be in the modified state. And it will be the responsibility of the processor to write back the modified value to block present in the main memory to ensure cache coherence.
- **Exclusive**: The block in the exclusive state indicate the clean copy of the block and the only copy of that block in any of the cache.
- **Shared**: The block will be in shared state when the block has been shared between two processors.
- **Invalid**: The block in an invalid state indicates that this block is not valid. The processor has to either get the updated/valid block from main memory or from processor which has the modified/updated block.

Transition Diagram:

Fig 5 shows the state transition diagram of the MESI. The state transition of the MESI is almost similar to the state transition of the MSI protocol. The only difference is the new state has been added which is called as the exclusive state.

If the block in the exclusive state indicates that the block is clean and it is the only copy present in any of the caches. The value of the block matches the value in the main memory. When the local-read operation is performed on the block which is present in the exclusive state, then the block will be in the exclusive state, but if we perform another operation like local-write on the block then the block will move to modify state. The operation like remote read will make the block to change its state from exclusive to shared state because the block will be shared between two processors and the for the remote-write operation the block will change its state to the invalid state because the other processor has modified the data in the block, so the data present in the block will be invalid.

	LR	LW	RR	RW
M(Modified)	M	M	S(WB)	I(WB)
E(Exclusive)	E	M	S	I
S(Shared)	S	M	S	I
I(Invalid)	S(READ MISS)	M(WRITE MISS)	I	I

Figure 6. State Transition Table for MESI Protocols

c. MOSI

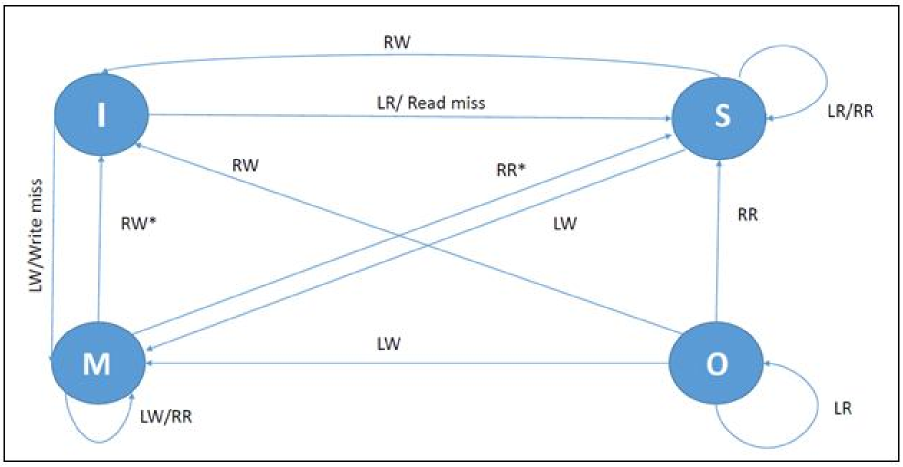


Figure 7. State Transition Diagram for MOSI Protocols

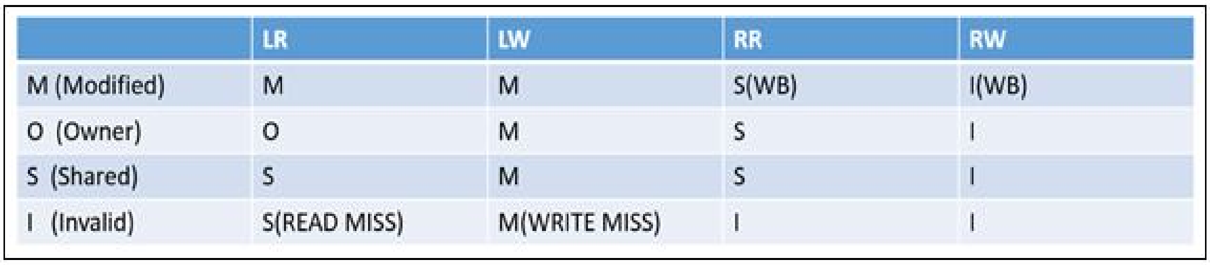
MOSI protocol is the extended version of MSI protocol. In MOSI protocol has the name clearly indicate it consists of four state:

- **Modified**: When the processor writes new value to the block then the block will be in the modified state. And it will be the responsibility of the processor to write back the modified value to block present in the main memory to ensure cache coherence.
- **Owned**: The block in the owned state indicate that the block has been owned by the processor. The block in the owned state hold the most recent and the correct copy of data.
- **Shared**: The block will be in shared state when the block has been shared between two processors.
- **Invalid**: The block in an invalid state indicates that this block is not valid. The processor has to either get the updated/valid block from main memory or from processor which has the modified/updated block.

Transition Diagram:
Consider the transition diagram in fig 7 which shows the state transition of the block. The state transition of the MOSI protocol is almost similar to the MSI protocol but the only difference is the MOSI protocol consists of an extra state which is Owned State.

The block in the owned state indicates that the block has been owned by the processor. So if any other processor request for this block they can fetch the block from the processor instead of the main memory this helps in reducing the overhead on the bus. Only one cache can hold a block in the owned state, other can have it in the shared state.

The operation like local-read on the block which is in owned state makes the block to stay in the same state, whereas the other operation like local-write make the block to move to modify state. The operation like remote-read on the block makes the block to move to shared state because the block has been shared between two processors and the remote-write operation makes the block to move to invalid state.



	LR	LW	RR	RW
M (Modified)	M	M	S(WB)	I(WB)
O (Owner)	O	M	S	I
S (Shared)	S	M	S	I
I (Invalid)	S(READ MISS)	M(WRITE MISS)	I	I

Figure 8. State Transition Table for MOSI Protocols

d. MERSI

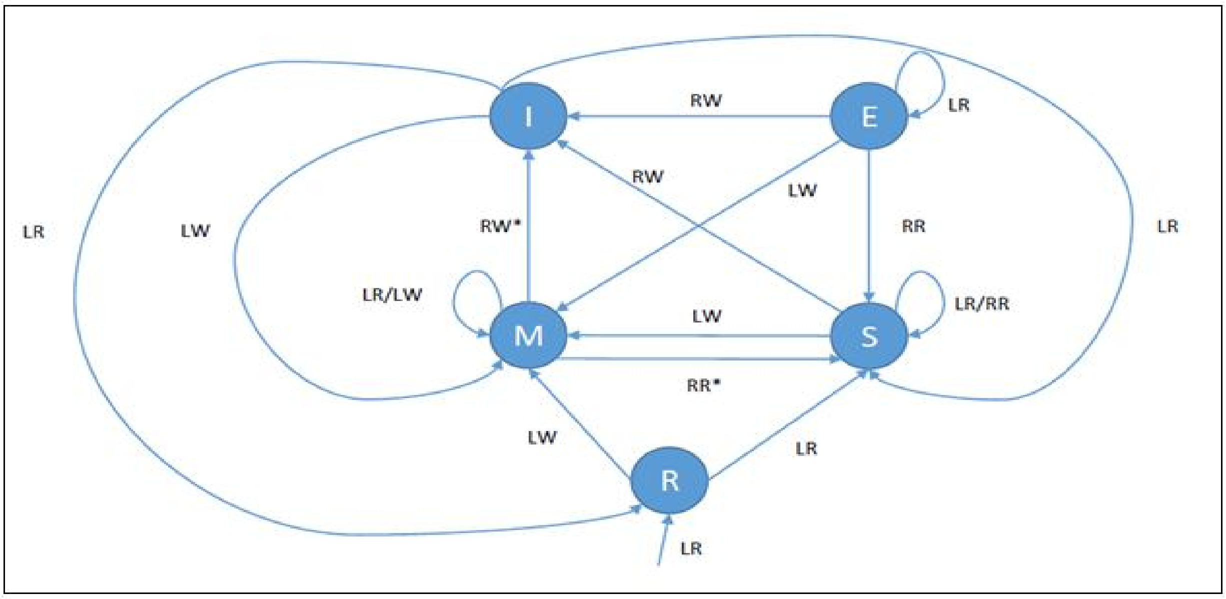


Figure 9. State Transition Diagram for MERSI Protocols

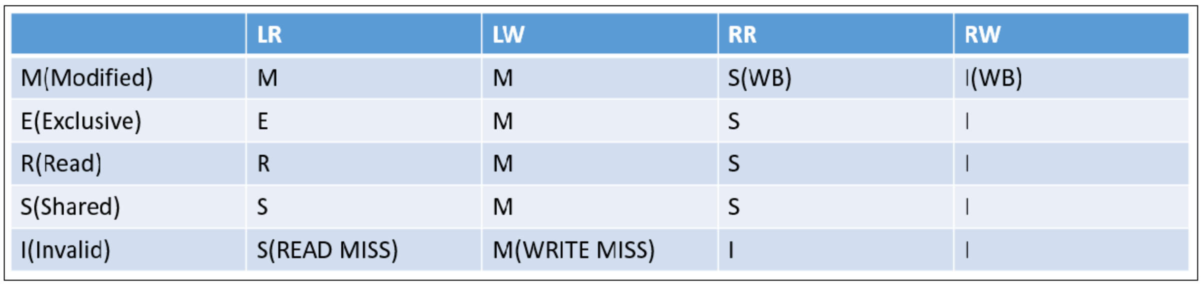
MERSI protocol is an extended version of MESI protocol with the new state which is read state. As the name indicates it consists of five state:

- **Modified**: When the processor writes new value to the block then the block will be in the modified state. And it will be the responsibility of the processor to write back the modified value to block present in the main memory to ensure cache coherence.
- **Exclusive**: The block in the exclusive state indicate the clean copy of the block and the only copy of that block in any of the cache.
- **Read**: This state is similar to the exclusive state. The block in read state is clean and valid. The processor has to request the owner ship of the block which is present in the read state before the processor modifies the block move to the modified state (M).
- **Shared**: The block will be in shared state when the block has been shared between two processors.
- **Invalid**: The block in an invalid state indicates that this block is not valid. The processor has to either get the updated/valid block from main memory or from processor which has the modified/updated block.

Transition Diagram:

Fig 9 shows the state transition diagram. The transition diagram is similar to MESI, but it consists of an extra state called as read state.

As we can see in Fig 9 if we want to perform local-write on the block which is in invalid state, then the processor has to get the permission from read state by performing local-read on block which is in invalid state and after performing above operation the block will change its state to read state and then by performing local-write, the block will be modified and then changes its state from read state to modify state.



	LR	LW	RR	RW
M(Modified)	M	M	S(WB)	I(WB)
E(Exclusive)	E	M	S	I
R(Read)	R	M	S	I
S(Shared)	S	M	S	I
I(Invalid)	S(READ MISS)	M(WRITE MISS)	I	I

Figure 10. State Transition Table for MERSI Protocols

2. <u>**Write Invalidate**</u>

In write-update the processor that has modified the data will broadcast the new data. All the cache which consists of copy of block in their local cache will be updated automatically. As all the blocks will be updated there is no need to the block to go to invalid state. The protocol that comes under write-update will not be having the invalid state. This is one of the advantage of write-update, but the disadvantage is the number of overhead on the bus will be increased.

a. DRAGON

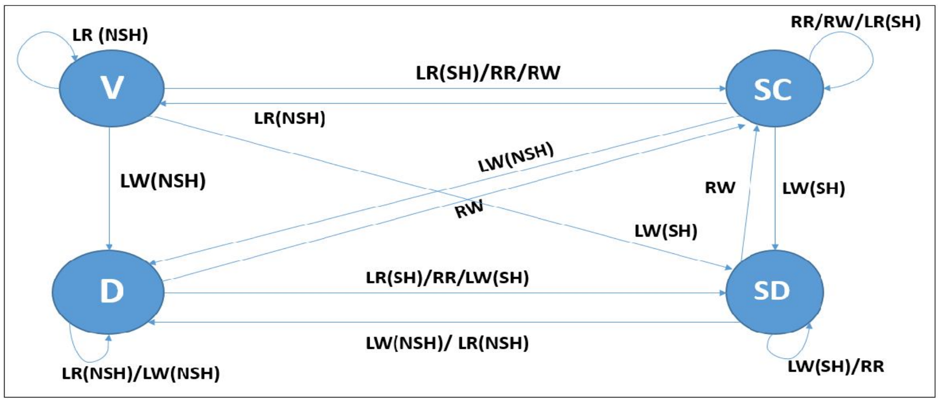


Figure 11. State Transition Diagram for DRAGON Protocols

In dragon protocol there are four states:

Valid: The block in the valid state indicates that it is clean and block is not shared between other processor. The block is present in the local cache of the processor and in the main memory.

Shared-Clean: The block in shared-clean state indicate that the given block is clean and block is shared with the other processor.

Dirty: When the block is modified then the block will go to Dirty state and the block is not shared between the other processors.

Shared-Dirty: If the block is in shared dirty it indicates that the block is modified and the block is shared with other processors. So the modified block will be updated to other processor which is having this modified block.

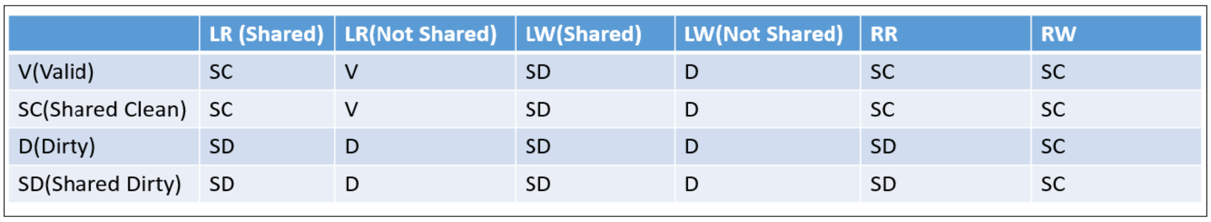
Transition Diagram:

Fig 11consists of state transition diagram of Dragon protocol. In Dragon protocol four new operation has been added. Local-read (shared), local-read (not shared), local-write (shared) and local-write (shared).

For the local-read (shared) all the block will be changed to shared-clean state because the block will be shared between the processors. But for the local-read (not shared) all the block will be changed to valid state because only one processor will be reading the block present in its known cache.

Similarly, for local-write(shared) the blocks will change its state to shared-dirty because if the processor has modified the block which is shared between the processors, then the modified block will be updated in the other processor which consists copy of the modified block. But for local-write (not shared) all the block will change its state to dirty because the modification is done to block which is not shared.

When the remote-write operation is performed on block which is any state make the block to change its state to shared-clean and for the block in dirty state when the remote-read operation is performed then the state of block will be changed from dirty state to shared-dirty.



	LR (Shared)	LR(Not Shared)	LW(Shared)	LW(Not Shared)	RR	RW
V(Valid)	SC	V	SD	D	SC	SC
SC(Shared Clean)	SC	V	SD	D	SC	SC
D(Dirty)	SD	D	SD	D	SD	SC
SD(Shared Dirty)	SD	D	SD	D	SD	SC

Figure 12. State Transition Table for DRAGON Protocols

b. FIREFLY

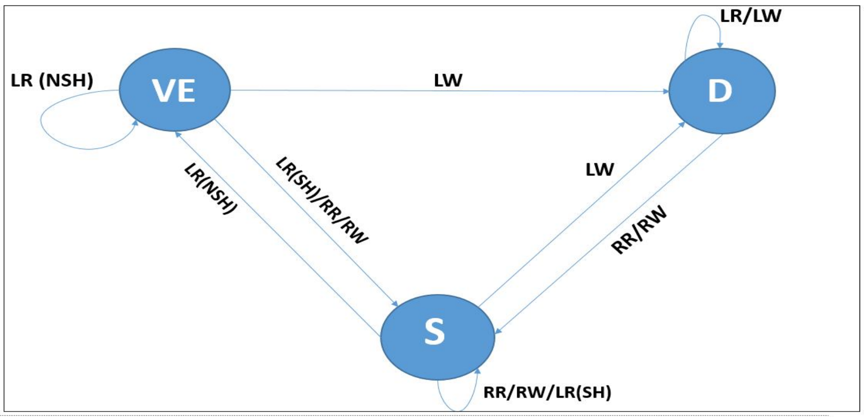


Figure 13. State Transition Diagram for FIREFLY Protocols

In firefly there are three states;

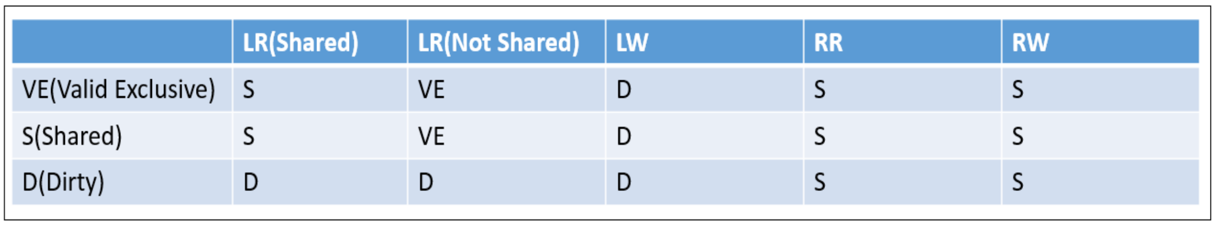
Valid-exclusive: The block in Valid-exclusive indiacte that this block has coherent copy of memory and this is the only copy present in the caches.

Shared: The block in the shared state indicate that the block has been shared between the processors.

Dirty: The block in dirty state indiacte that the block has been modified and the blcok will be written back to the memory at replacement.

Transition Diagram:

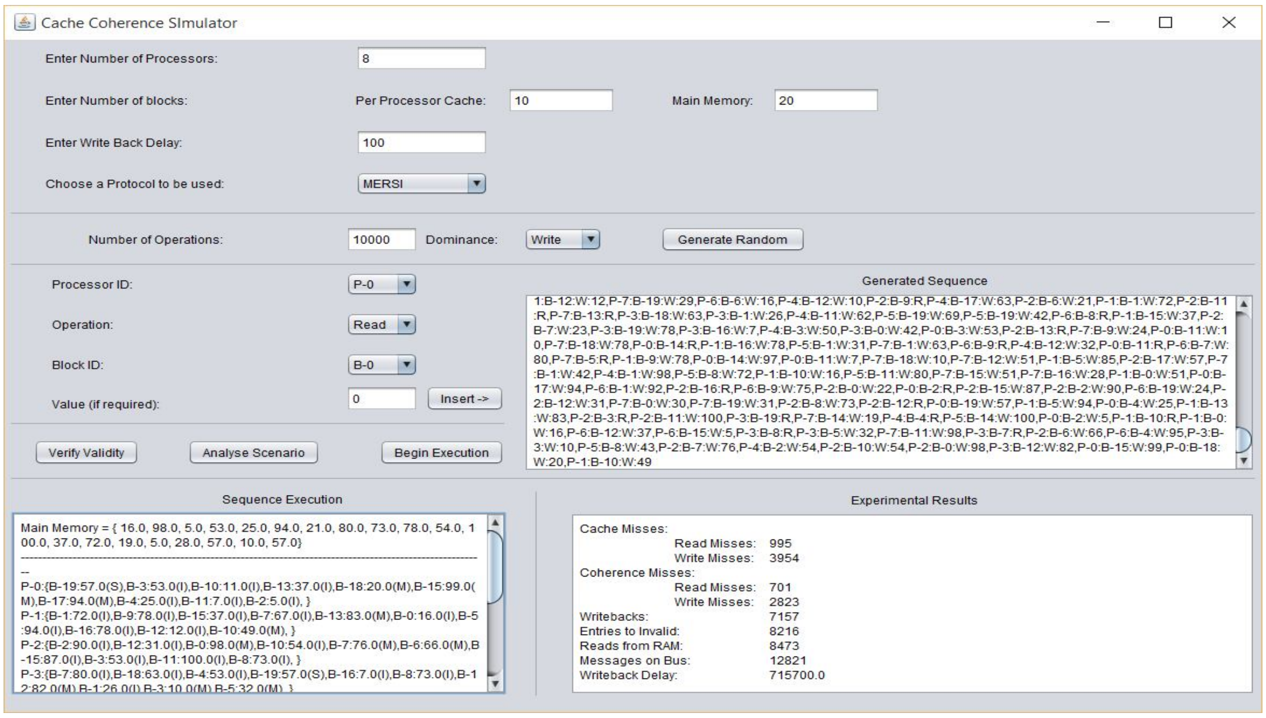
Fig 13 shows the state transition diagram of the block in the firfly protocol. It consists of three state valid-exclusive, dirty and shared. In firefly protocol two new operation is used which are local-read(shared) and local-read(not shared). When the operation like remote-read, remote-wite and local-read(shared) is perfromed on the block which is in any state makes the block to change is state to shared state because the block will be shared between the processors. For the operation like local-read(not shared) performed on the block in any state make the block change its state to the valid-exclusive. But for the operation like local-write makes the block change its state to the dirty block.



	LR(Shared)	LR(Not Shared)	LW	RR	RW
VE(Valid Exclusive)	S	VE	D	S	S
S(Shared)	S	VE	D	S	S
D(Dirty)	D	D	D	S	S

Figure 14. State Transition Table for FIREFLY Protocols

III. Simulator Implementation


Cache Coherence Simulator

Enter Number of Processors: 8

Enter Number of blocks: Per Processor Cache: 10 Main Memory: 20

Enter Write Back Delay: 100

Choose a Protocol to be used: MERSI

Number of Operations: 10000 Dominance: Write

Processor ID: P-0

Operation: Read

Block ID: B-0

Value (if required): 0

Generated Sequence

```
1:B-12:W:12,P-7:B-19:W:29,P-6:B-6:W:16,P-4:B-12:W:10,P-2:B-9:R,P-4:B-17:W:63,P-2:B-6:W:21,P-1:B-1:W:72,P-2:B-11
:R,P-7:B-13:R,P-3:B-18:W:63,P-3:B-1:W:26,P-4:B-11:W:62,P-5:B-19:W:69,P-5:B-19:W:42,P-6:B-8:R,P-1:B-15:W:37,P-2:
B-7:W:23,P-3:B-19:W:78,P-3:B-16:W:7,P-4:B-3:W:50,P-3:B-0:W:42,P-0:B-3:W:53,P-2:B-13:R,P-7:B-9:W:24,P-0:B-11:W:1
0,P-7:B-18:W:78,P-0:B-14:R,P-1:B-16:W:78,P-5:B-1:W:31,P-7:B-1:W:63,P-6:B-9:R,P-4:B-12:W:32,P-0:B-11:R,P-6:B-7:W:
80,P-7:B-5:R,P-1:B-9:W:78,P-0:B-14:W:97,P-0:B-11:W:7,P-7:B-18:W:10,P-7:B-12:W:51,P-1:B-5:W:85,P-2:B-17:W:57,P-7
:B-1:W:42,P-4:B-1:W:98,P-5:B-8:W:72,P-1:B-10:W:16,P-5:B-11:W:80,P-7:B-15:W:51,P-7:B-16:W:28,P-1:B-0:W:51,P-0:B-
17:W:94,P-6:B-1:W:92,P-2:B-16:R,P-6:B-9:W:75,P-2:B-0:W:22,P-0:B-2:R,P-2:B-15:W:87,P-2:B-2:W:90,P-6:B-19:W:24,P-
2:B-12:W:31,P-7:B-0:W:30,P-7:B-19:W:31,P-2:B-8:W:73,P-2:B-12:R,P-0:B-19:W:57,P-1:B-5:W:94,P-0:B-4:W:25,P-1:B-13
:W:83,P-2:B-3:R,P-2:B-11:W:100,P-3:B-19:R,P-7:B-14:W:19,P-4:B-4:R,P-5:B-14:W:100,P-0:B-2:W:5,P-1:B-10:R,P-1:B-0:
W:16,P-6:B-12:W:37,P-6:B-15:W:5,P-3:B-8:R,P-3:B-5:W:32,P-7:B-11:W:98,P-3:B-7:R,P-2:B-6:W:66,P-6:B-4:W:95,P-3:B-
3:W:10,P-5:B-8:W:43,P-2:B-7:W:76,P-4:B-2:W:54,P-2:B-10:W:54,P-2:B-0:W:98,P-3:B-12:W:82,P-0:B-15:W:99,P-0:B-18:
W:20,P-1:B-10:W:49
```

Sequence Execution

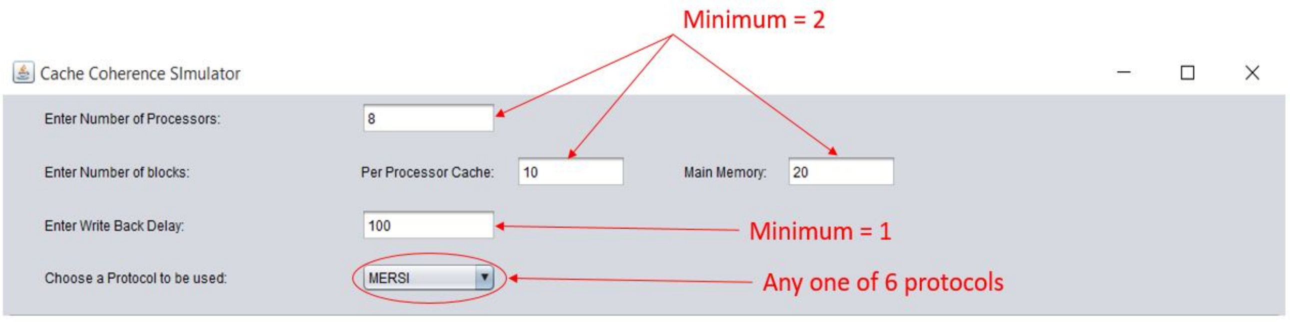
```
Main Memory = { 16.0, 98.0, 5.0, 53.0, 25.0, 94.0, 21.0, 80.0, 73.0, 78.0, 54.0, 1
00.0, 37.0, 72.0, 19.0, 5.0, 28.0, 57.0, 10.0, 57.0}
--------------------------------------------------------------------------------
--
P-0:{B-19:57.0(S),B-3:53.0(I),B-10:11.0(I),B-13:37.0(I),B-18:20.0(M),B-15:99.0(
M),B-17:94.0(M),B-4:25.0(I),B-11:7.0(I),B-2:5.0(I), }
P-1:{B-1:72.0(I),B-9:78.0(I),B-15:37.0(I),B-7:67.0(I),B-13:83.0(M),B-0:16.0(I),B-5
:94.0(I),B-16:78.0(I),B-12:12.0(I),B-10:49.0(M), }
P-2:{B-2:90.0(I),B-12:31.0(I),B-0:98.0(M),B-10:54.0(I),B-7:76.0(M),B-6:66.0(M),B
-15:87.0(I),B-3:53.0(I),B-11:100.0(I),B-8:73.0(I), }
P-3:{B-7:80.0(I),B-18:63.0(I),B-4:53.0(I),B-19:57.0(S),B-16:7.0(I),B-8:73.0(I),B-1
2:82.0(M),B-1:26.0(I),B-3:10.0(M),B-5:32.0(M), }
```

Experimental Results

```
Cache Misses:
            Read Misses:    995
            Write Misses:   3954
Coherence Misses:
            Read Misses:    701
            Write Misses:   2823
Writebacks:                 7157
Entries to Invalid:         8216
Reads from RAM:             8473
Messages on Bus:            12821
Writeback Delay:            715700.0
```

Figure 15. Simulator Snapshot

We started off implementation of this simulator by building the central engine that drives the multiprocessor architecture. This multiprocessor architecture is dependent on the input user provides. As seen in figure 16, there are several inputs required to be specified:


Cache Coherence Simulator

Enter Number of Processors: 8

Enter Number of blocks: Per Processor Cache: 10 Main Memory: 20

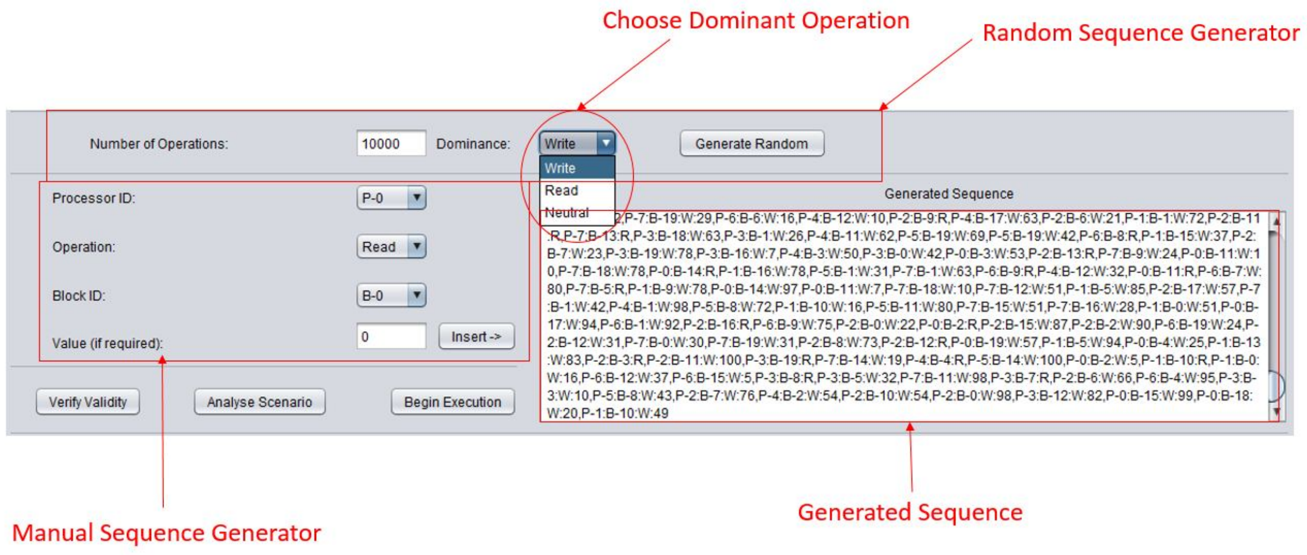
Enter Write Back Delay: 100

Choose a Protocol to be used: MERSI

Figure 16. Simulator Input Parameters

Number of processors define the number of processors and in turn the number of dedicated caches. As we are dealing with a multiprocessor architecture, the minimum requirement for a value entered in number of processors is 2. Similarly, the user needs to define how many blocks local dedicated cache is capable of accommodating as well as the number of blocks that the main memory holds. We also provide the user with the capability defining the writeback delay of the system. This delay corresponds to the delay a processor will face in order to write an updated value back into main memory/low level cache. The user may choose any unit for this value as it will not make a difference as long as it is kept consistent across experimentation.

Next part of the simulator focuses on generating sequence of operations on which the architecture will evaluate the chosen cache coherence enforcement protocol. There are three ways in which a user can specify a sequence for evaluation.



Number of Operations: 10000
Dominance: Write

Processor ID: P-0
Operation: Read
Block ID: B-0
Value (if required): 0

Generated Sequence

```
2,P-7:B-19:W:29,P-6:B-6:W:16,P-4:B-12:W:10,P-2:B-9:R,P-4:B-17:W:63,P-2:B-6:W:21,P-1:B-1:W:72,P-2:B-11
:R,P-7:B-13:R,P-3:B-18:W:63,P-3:B-1:W:26,P-4:B-11:W:62,P-5:B-19:W:69,P-5:B-19:W:42,P-6:B-8:R,P-1:B-15:W:37,P-2:
B-7:W:23,P-3:B-19:W:78,P-3:B-16:W:7,P-4:B-3:W:50,P-3:B-0:W:42,P-0:B-3:W:53,P-2:B-13:R,P-7:B-9:W:24,P-0:B-11:W:1
0,P-7:B-18:W:78,P-0:B-14:R,P-1:B-16:W:78,P-5:B-1:W:31,P-7:B-1:W:63,P-6:B-9:R,P-4:B-12:W:32,P-0:B-11:R,P-6:B-7:W:
80,P-7:B-5:R,P-1:B-9:W:78,P-0:B-14:W:97,P-0:B-11:W:7,P-7:B-18:W:10,P-7:B-12:W:51,P-1:B-5:W:85,P-2:B-17:W:57,P-7
:B-1:W:42,P-4:B-1:W:98,P-5:B-8:W:72,P-1:B-10:W:16,P-5:B-11:W:80,P-7:B-15:W:51,P-7:B-16:W:28,P-1:B-0:W:51,P-0:B-
17:W:94,P-6:B-1:W:92,P-2:B-16:R,P-6:B-9:W:75,P-2:B-0:W:22,P-0:B-2:R,P-2:B-15:W:87,P-2:B-2:W:90,P-6:B-19:W:24,P-
2:B-12:W:31,P-7:B-0:W:30,P-7:B-19:W:31,P-2:B-8:W:73,P-2:B-12:R,P-0:B-19:W:57,P-1:B-5:W:94,P-0:B-4:W:25,P-1:B-13
:W:83,P-2:B-3:R,P-2:B-11:W:100,P-3:B-19:R,P-7:B-14:W:19,P-4:B-4:R,P-5:B-14:W:100,P-0:B-2:W:5,P-1:B-10:R,P-1:B-0:
W:16,P-6:B-12:W:37,P-6:B-15:W:5,P-3:B-8:R,P-3:B-5:W:32,P-7:B-11:W:98,P-3:B-7:R,P-2:B-6:W:66,P-6:B-4:W:95,P-3:B-
3:W:10,P-5:B-8:W:43,P-2:B-7:W:76,P-4:B-2:W:54,P-2:B-10:W:54,P-2:B-0:W:98,P-3:B-12:W:82,P-0:B-15:W:99,P-0:B-18:
W:20,P-1:B-10:W:49
```

Figure 17. Simulator Sequence Generation

As seen in figure 17, the first way to enter in a sequence is to generate it using the random sequence generator. This generator allows you to specify the number of operations and which operation should be considered as the dominant operation while generating this sequence. Following algorithm illustrates how the dominance of an operation is incorporated while generating the sequence.

```
Function getNextOperation()
    n = getRandomNumberBetweenZeroAndOne();

    switch(dominance)
        case WriteDominant:
            if(n < 0.2) return READ;
            else return WRITE;
        case ReadDominant:
            if(n < 0.2) return WRITE;
            else return READ;
        case NeutralDominant:
            if(n < 0.5) return READ;
            else return WRITE;
End function
```

The other two ways to specify a sequence are pretty straightforward. One being that the user can manually type in the sequence that they want executed. We provide the user with the capability to verify that the sequence that they have entered is valid or not. Lastly, the user can use the manual sequence generator to add one operation at a time.

Once the user hits begin, the execution process begins. Instead of rambling on here about how this long process is carried out, we will present this in the form of algorithms as follows:

```
Function beginExecution()
    For every operation O by processor P
        If O is read,
            For every processor Q other than P
                Q.remoteRead(P);
            End for
            P.localRead();
        Else
            For every processor Q other than P
                Q.remoteWrite(P);
            End for
            P.localWrite();
        End if
    End for
End function
```

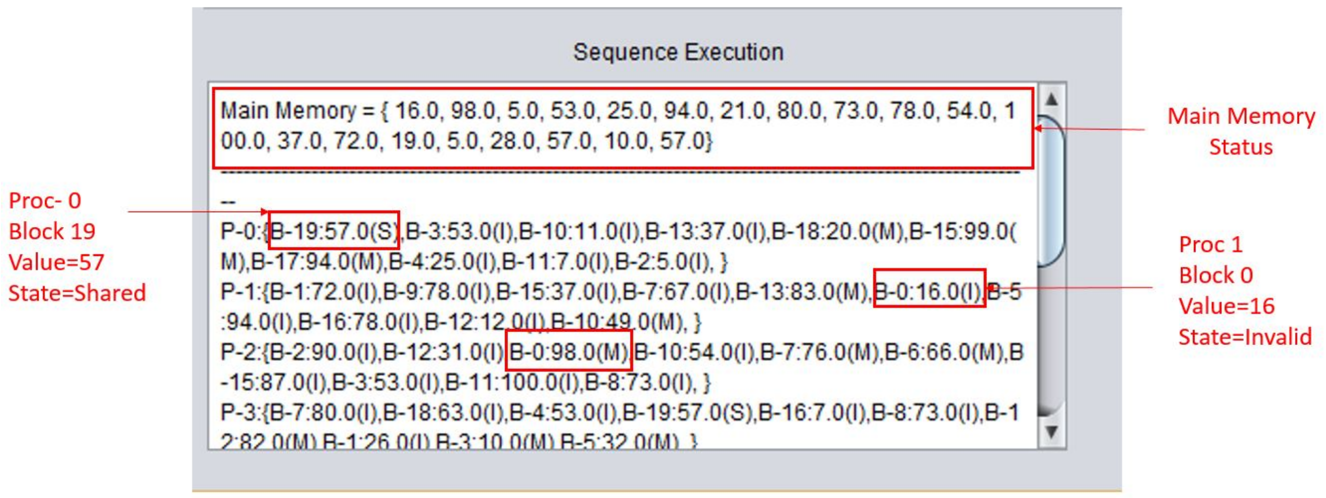
```
Function Protocol.remoteRead()
    if blockIsInMemory
        if blockIsDirty
            Perform Writeback or Share block (depending on protocol);
        end if
        Update state of block;
    End if
End function
```

```
Function Protocol.localRead()
    if NOT blockIsInMemory
        bring block into memory;
    End if
    Update state of block;
End function
```

```
Function Protocol.remoteWrite()
    if blockIsInMemory
        if blockIsDirty
            Perform Writeback or Share block (depending on protocol);
        end if
        Update state of block;
    End if
End function
```

```
Function Protocol.localWrite()
    if NOT blockIsInMemory
        bring block into memory;
    End if
    Update state and value of block;
End function
```

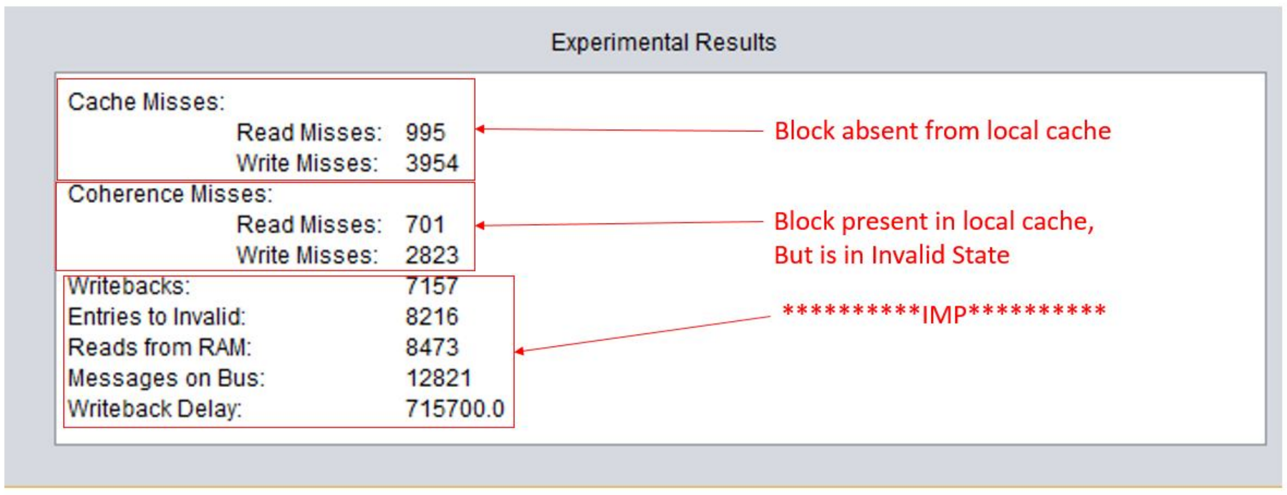
While the process of execution goes on, the user gets to see the states of each block being updated along with its values in the following block.



```
Main Memory = { 16.0, 98.0, 5.0, 53.0, 25.0, 94.0, 21.0, 80.0, 73.0, 78.0, 54.0, 1
00.0, 37.0, 72.0, 19.0, 5.0, 28.0, 57.0, 10.0, 57.0}
------------------------------------------------------------------------------------
--
P-0:{B-19:57.0(S),B-3:53.0(I),B-10:11.0(I),B-13:37.0(I),B-18:20.0(M),B-15:99.0(
M),B-17:94.0(M),B-4:25.0(I),B-11:7.0(I),B-2:5.0(I), }
P-1:{B-1:72.0(I),B-9:78.0(I),B-15:37.0(I),B-7:67.0(I),B-13:83.0(M),B-0:16.0(I),B-5
:94.0(I),B-16:78.0(I),B-12:12.0(I),B-10:49.0(M), }
P-2:{B-2:90.0(I),B-12:31.0(I),B-0:98.0(M),B-10:54.0(I),B-7:76.0(M),B-6:66.0(M),B
-15:87.0(I),B-3:53.0(I),B-11:100.0(I),B-8:73.0(I), }
P-3:{B-7:80.0(I),B-18:63.0(I),B-4:53.0(I),B-19:57.0(S),B-16:7.0(I),B-8:73.0(I),B-1
2:82.0(M),B-1:26.0(I),B-3:10.0(M),B-5:32.0(M), }
```

Figure 18. Simulator Execution View

Each operation is executed with a 500 ms delay so that the user can see how each operation has an impact on the state of all the blocks involved. Once the execution completes, the evaluation parameters are displayed in the block shown in figure 19.

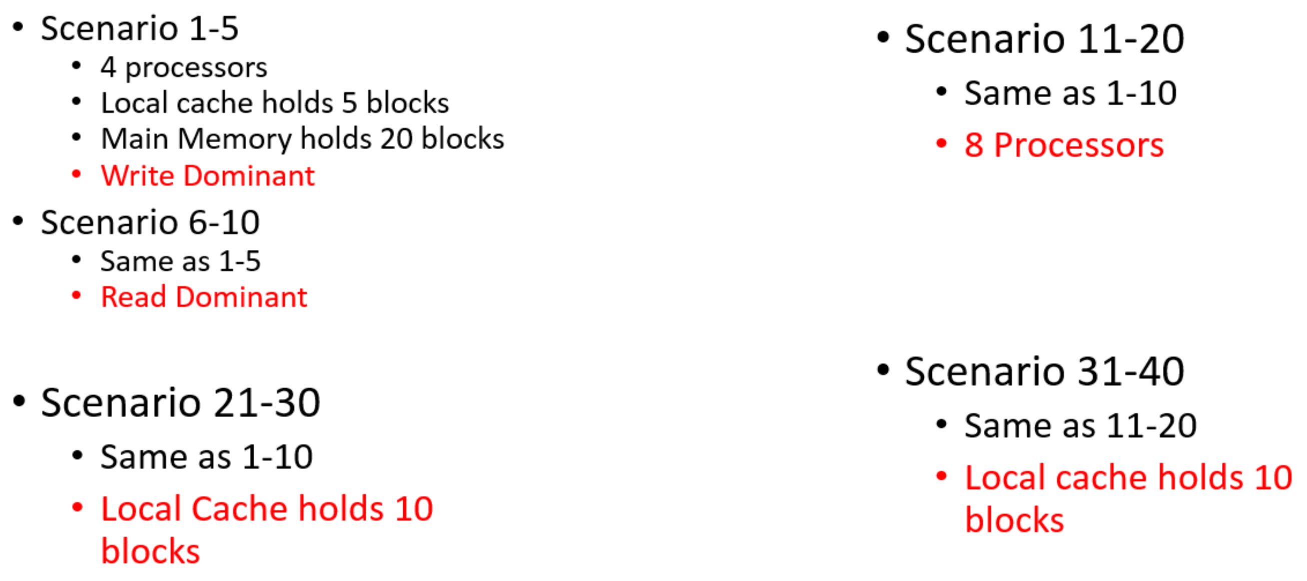


```
Cache Misses:
                Read Misses:    995
                Write Misses:   3954
Coherence Misses:
                Read Misses:    701
                Write Misses:   2823
Writebacks:                     7157
Entries to Invalid:             8216
Reads from RAM:                 8473
Messages on Bus:                12821
Writeback Delay:                715700.0
```

Figure 19. Simulator Evaluation Parameters

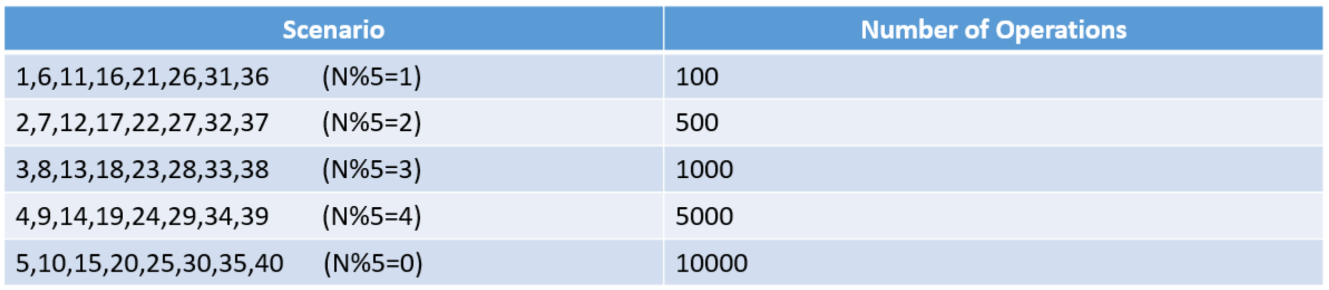
IV. Empirical Evaluation

In this section, we present experimental results that we performed using defined scenarios. Details of the scenarios are provided in the following figure:



- Scenario 1-5
  - 4 processors
  - Local cache holds 5 blocks
  - Main Memory holds 20 blocks
  - Write Dominant
- Scenario 6-10
  - Same as 1-5
  - Read Dominant
- Scenario 11-20
  - Same as 1-10
  - 8 Processors
- Scenario 21-30
  - Same as 1-10
  - Local Cache holds 10 blocks
- Scenario 31-40
  - Same as 11-20
  - Local cache holds 10 blocks

(a)



Scenario	Number of Operations
1,6,11,16,21,26,31,36 (N%5=1)	100
2,7,12,17,22,27,32,37 (N%5=2)	500
3,8,13,18,23,28,33,38 (N%5=3)	1000
4,9,14,19,24,29,34,39 (N%5=4)	5000
5,10,15,20,25,30,35,40 (N%5=0)	10000

(b)

Figure 20. (a) Scenario Differences (b) Length of sequence in each scenario

Now we take a look at experimental evaluation for each parameter using all scenarios. First let us look at cache misses and coherence misses observed across all scenarios.

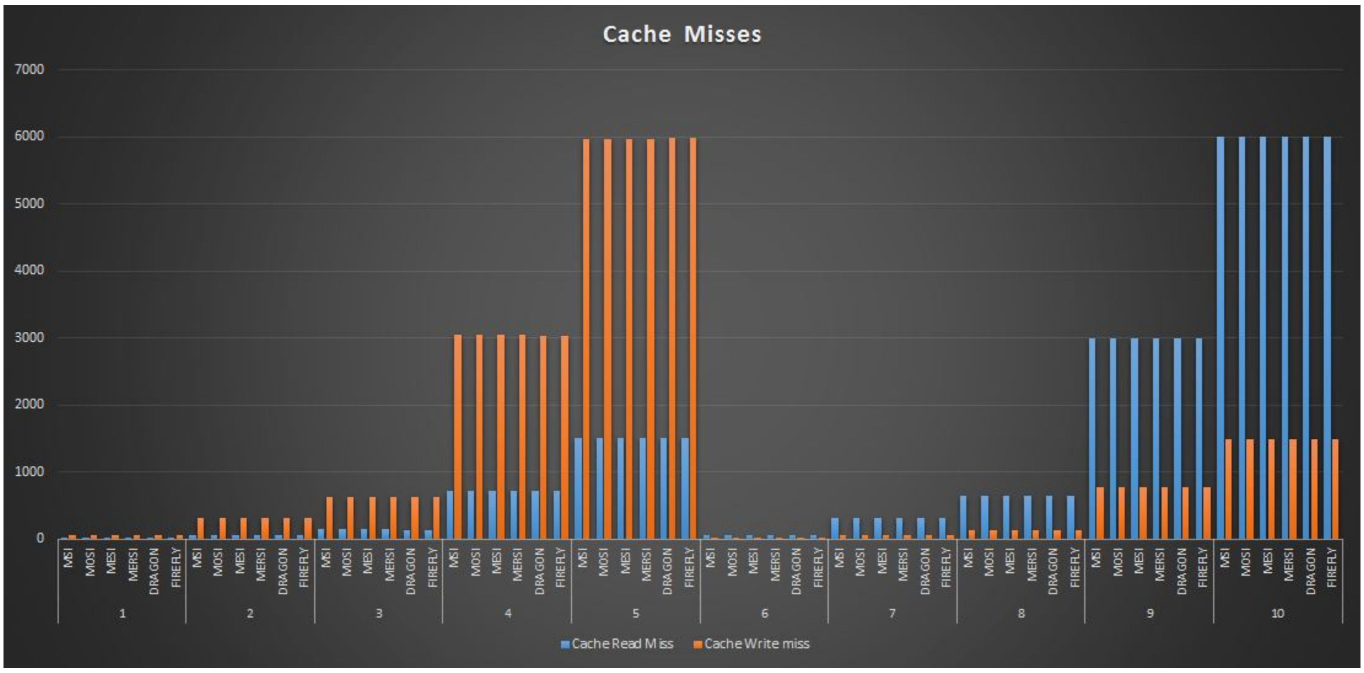


Figure 21. Cache Misses Scenario 1-10

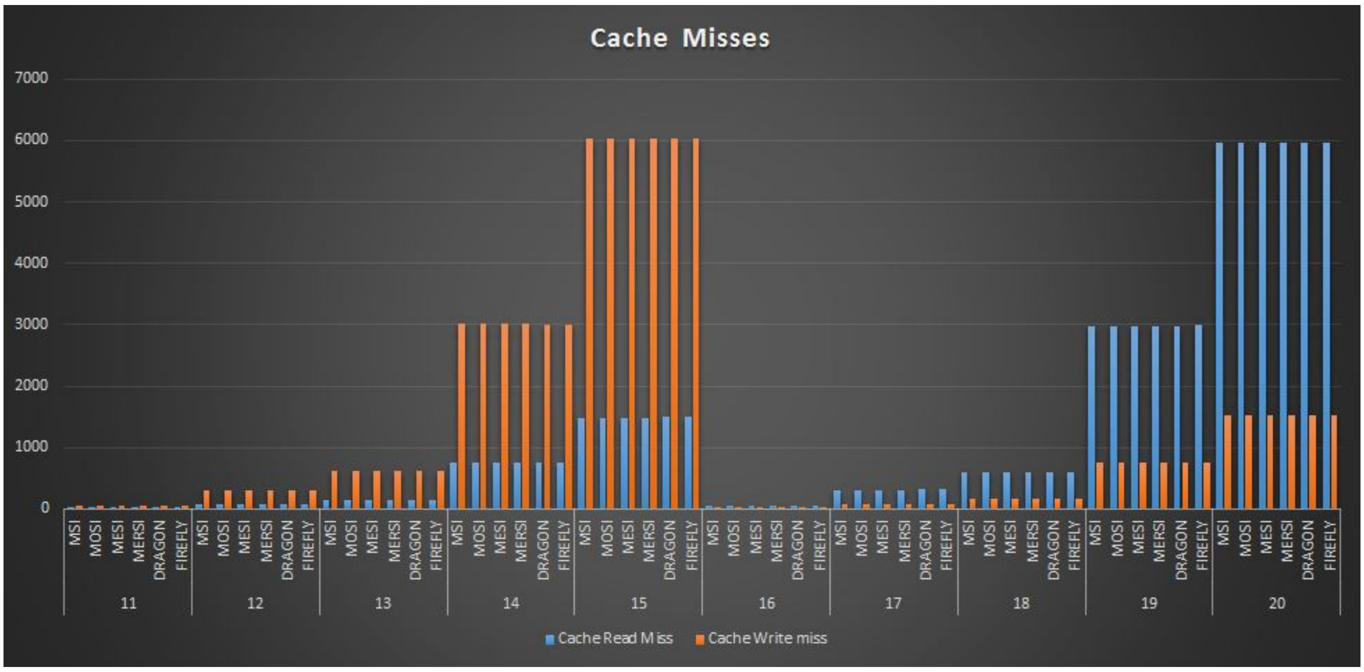


Figure 22. Cache Misses Scenario 11-20

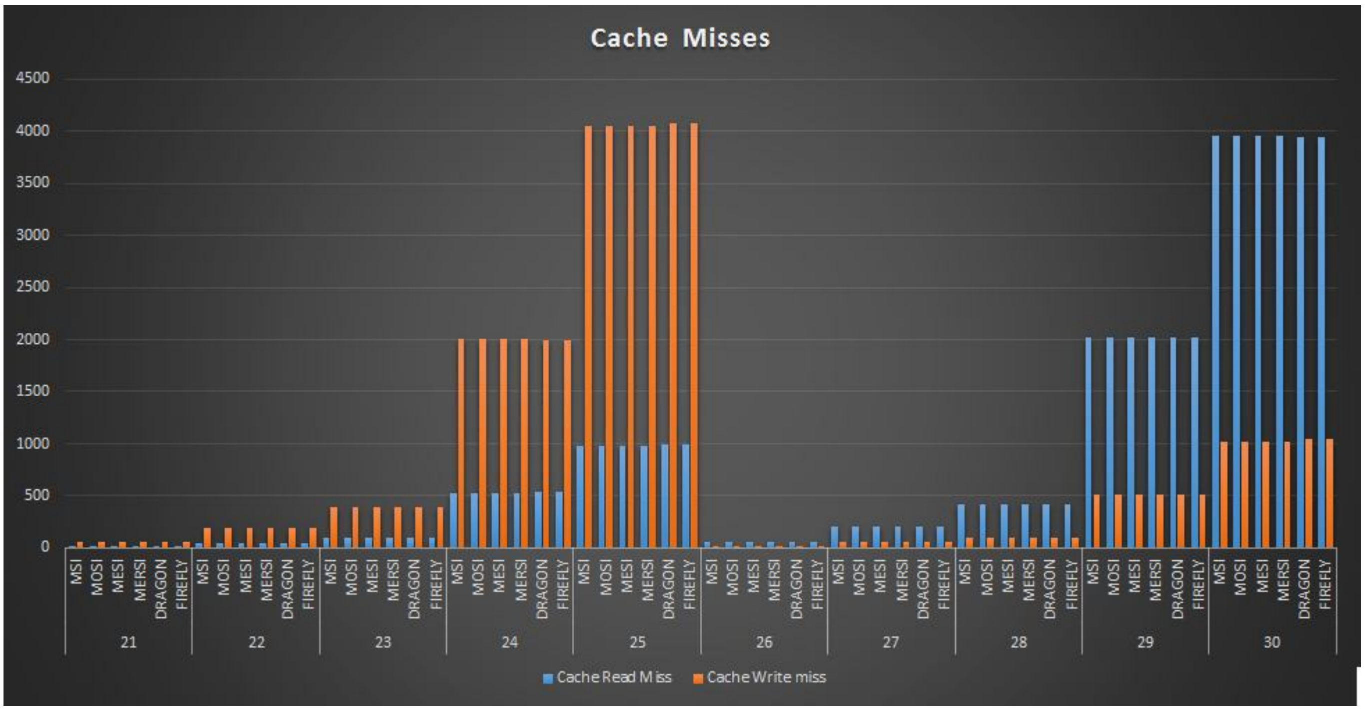


Figure 23. Cache Misses Scenario 21-30

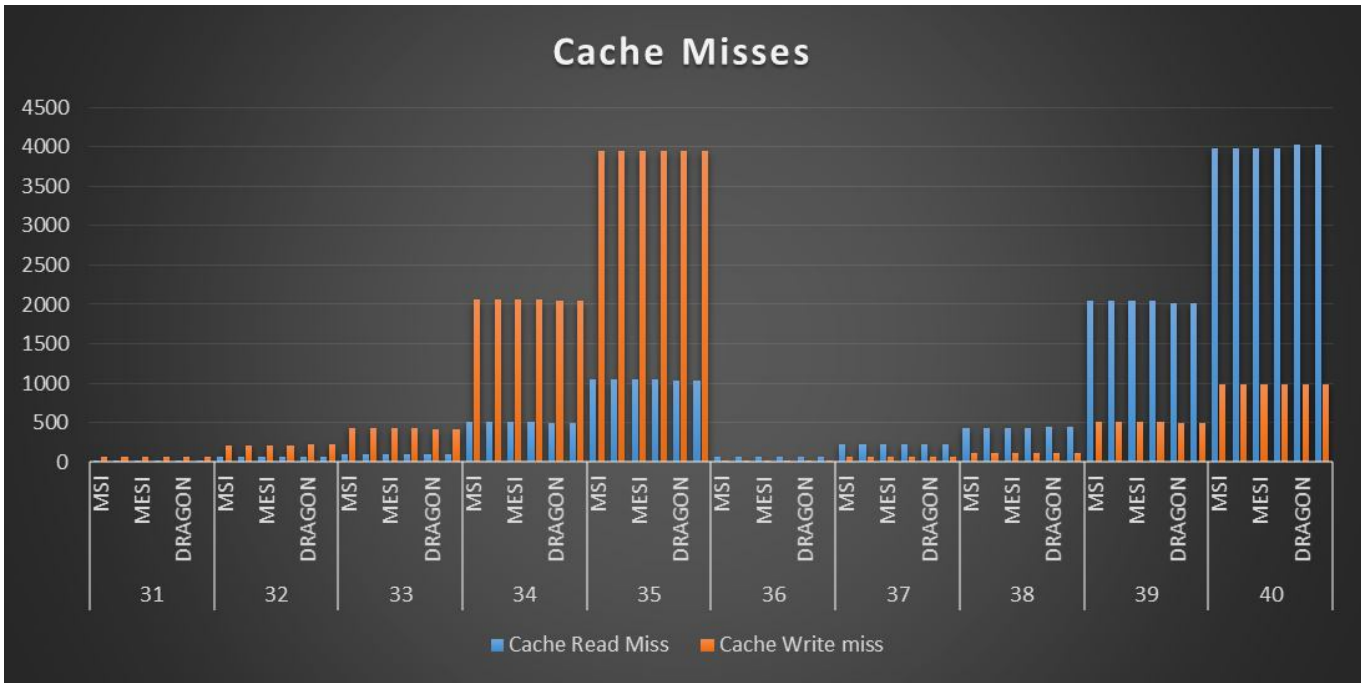


Figure 24. Cache Misses Scanrio 31-40

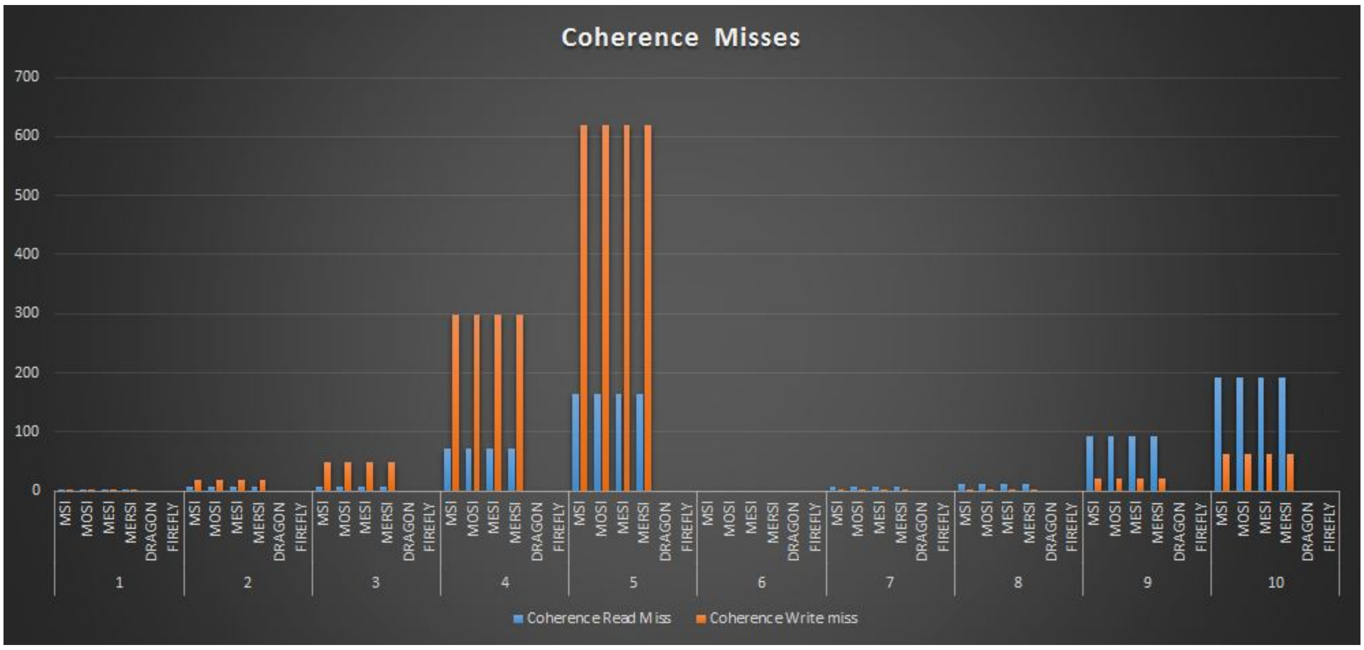


Figure 25. Coherence Misses Scenario 1-10

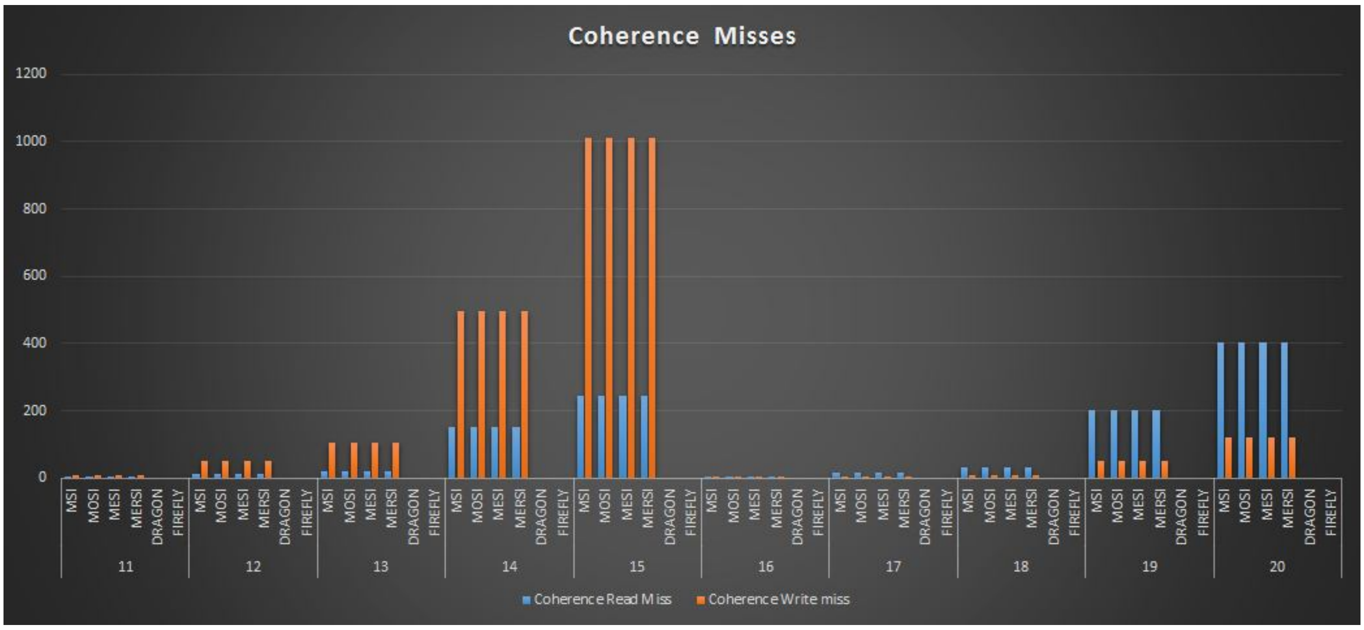


Figure 26. Coherence Misses Scenario 11-20

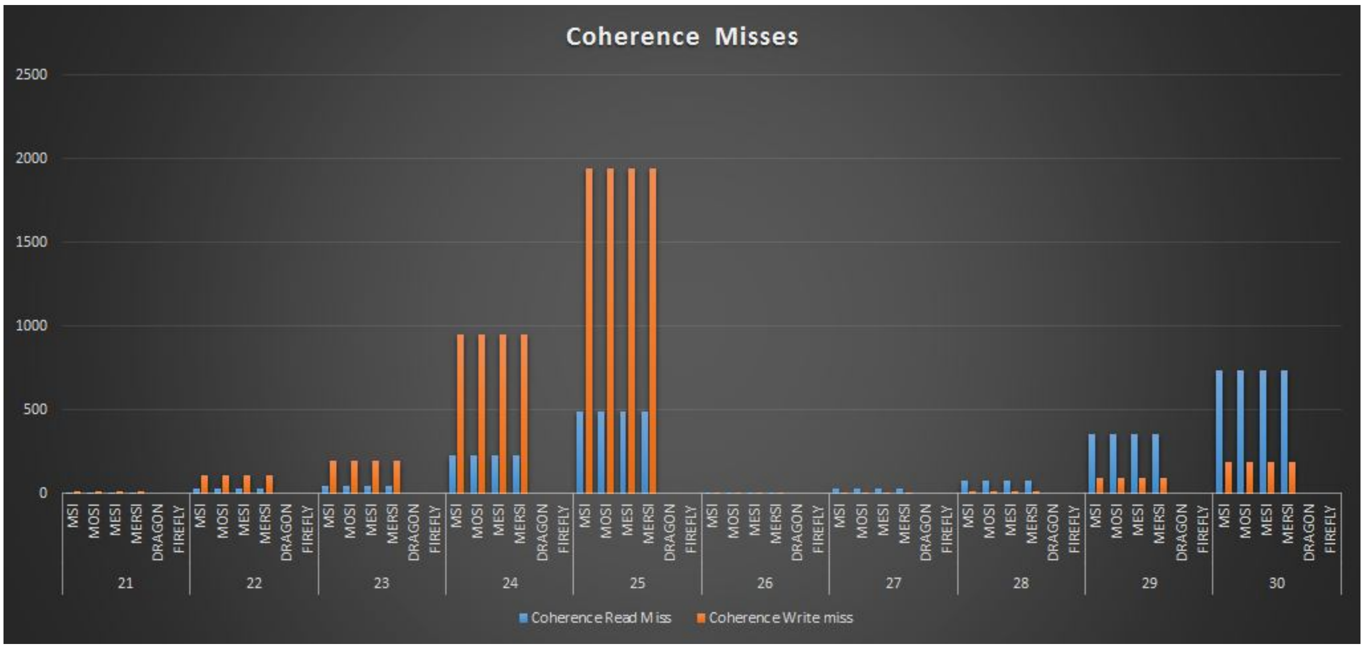


Figure 27. Coherence Misses Scenario 21-30

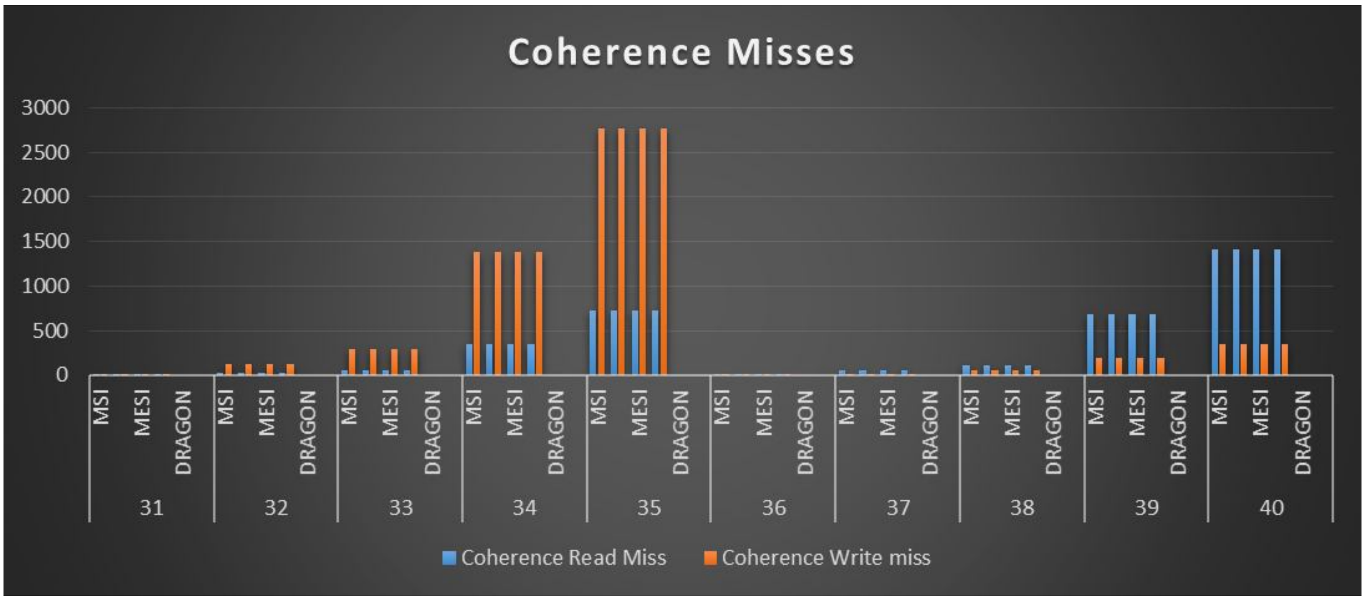


Figure 28. Coherence Misses Scenario 31-40

Let us now take a look at some of the other parameters which we evaluated the protocols on.

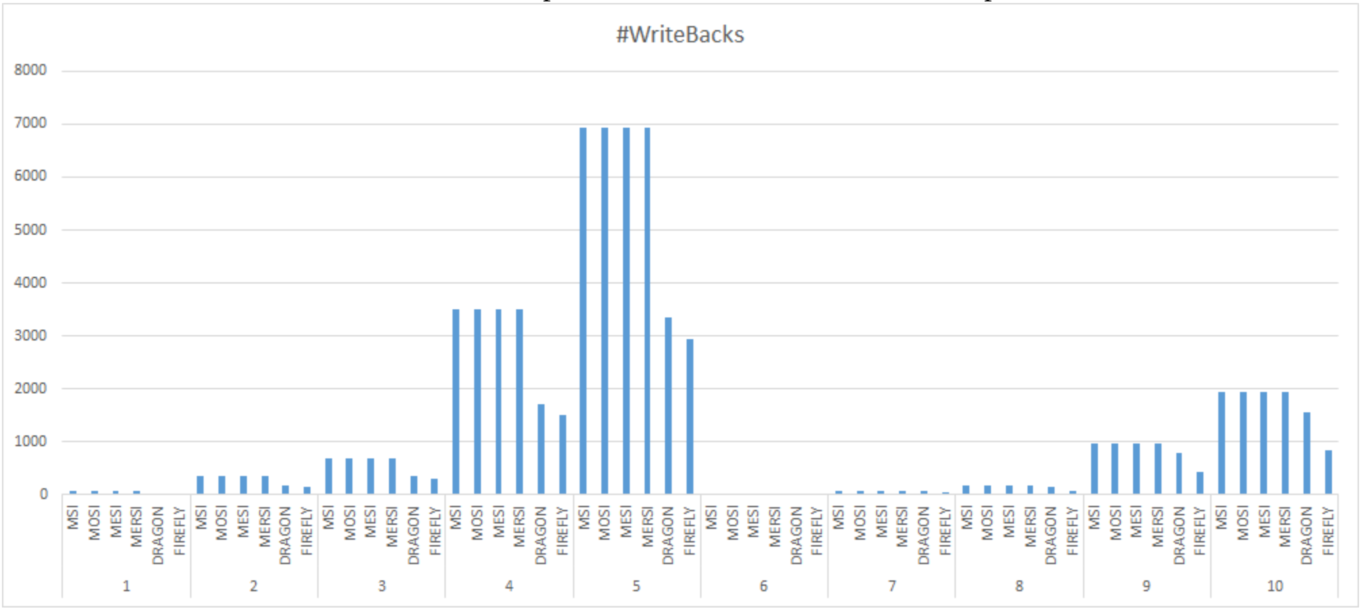


Figure 29. Writebacks Scenario 1-10

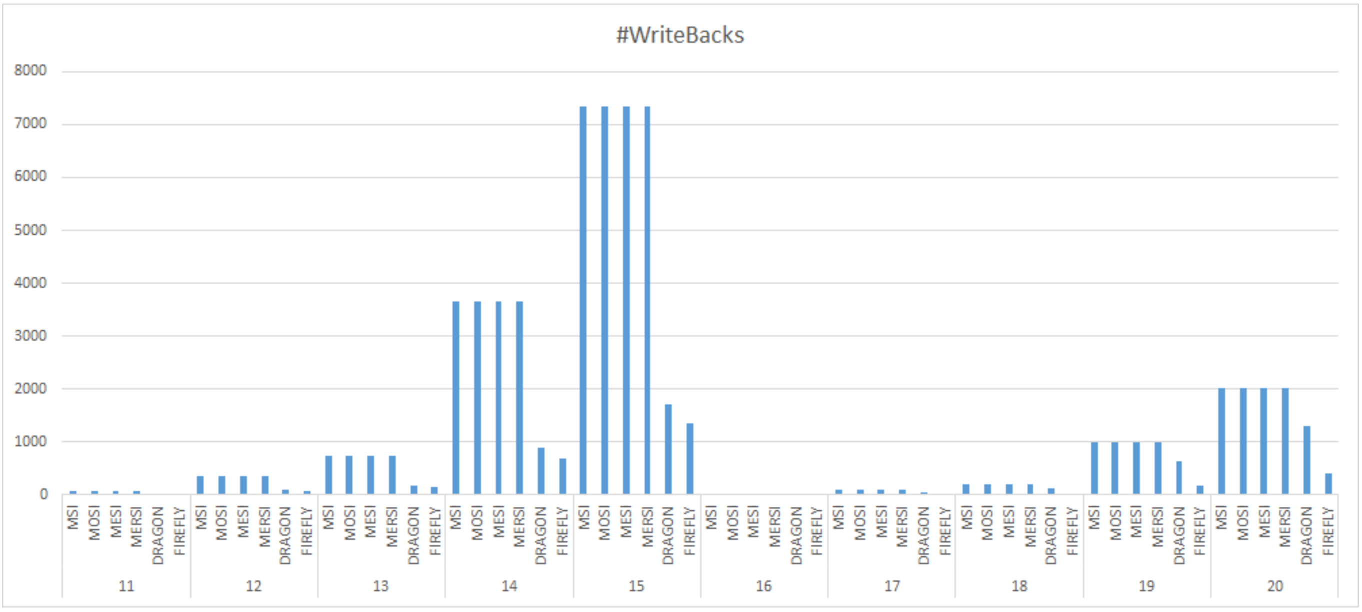


Figure 30. Writebacks Scenario 11-20

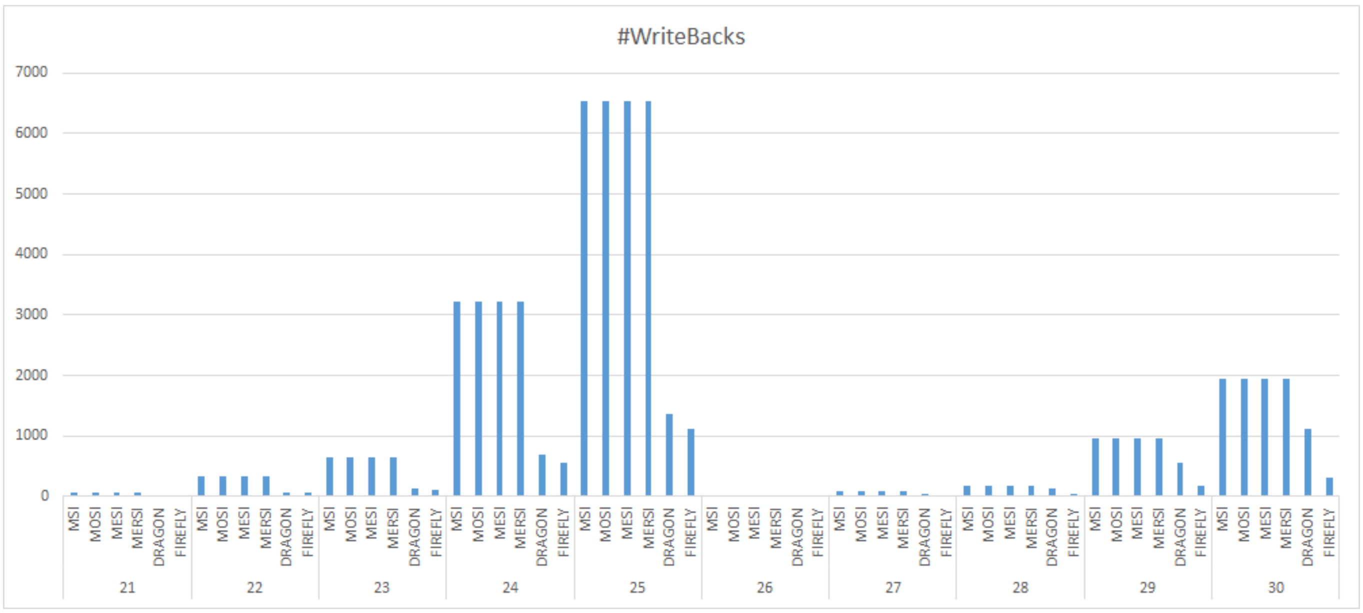


Figure 31. Writebacks Scenario 21-30

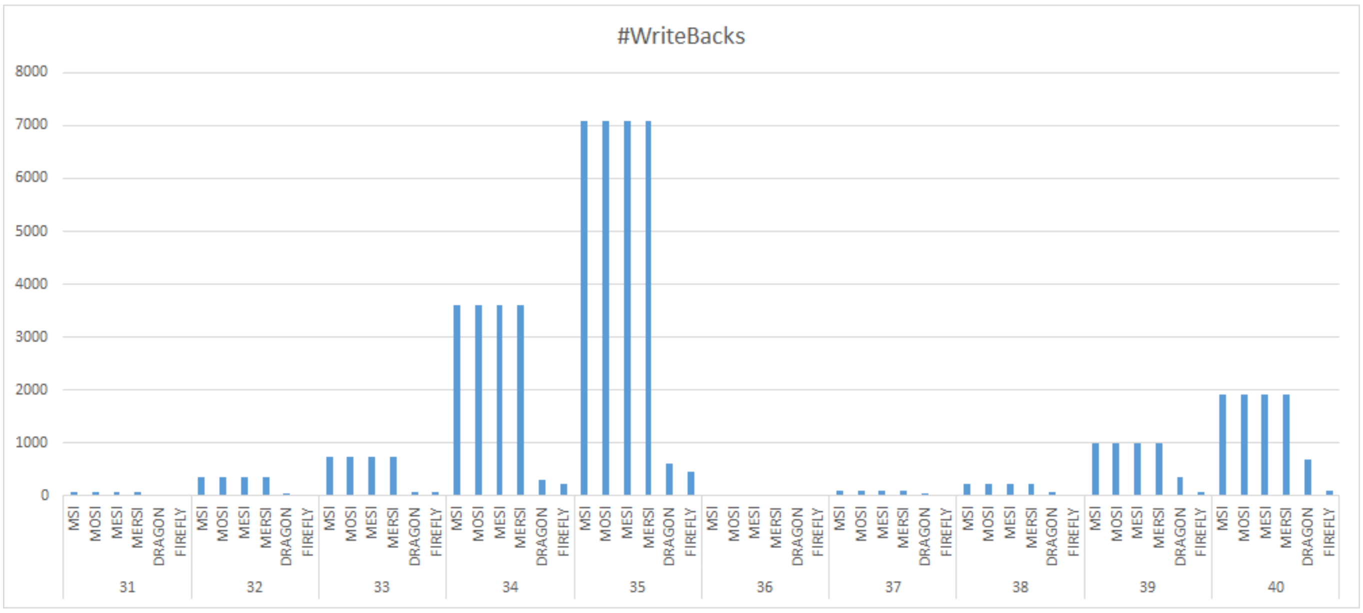


Figure 32. Writebacks Scenario 31-40

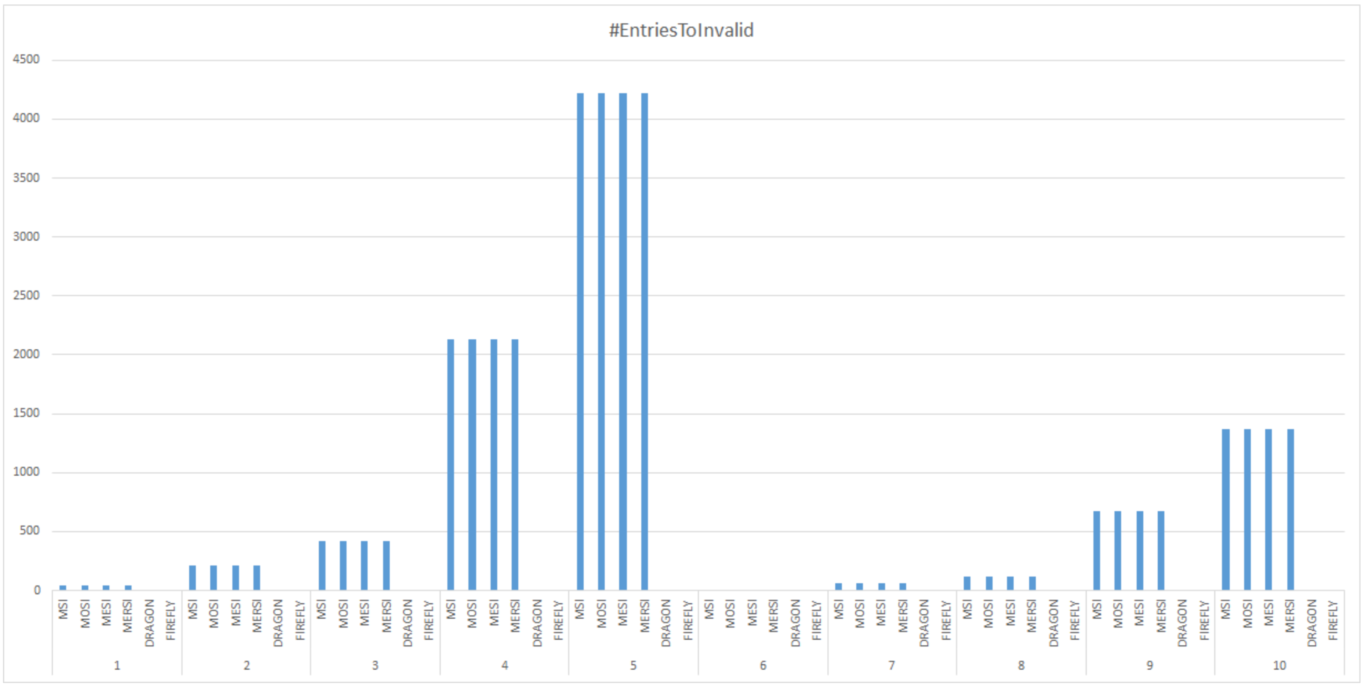


Figure 33. Entries to invalid Scenario 1-10

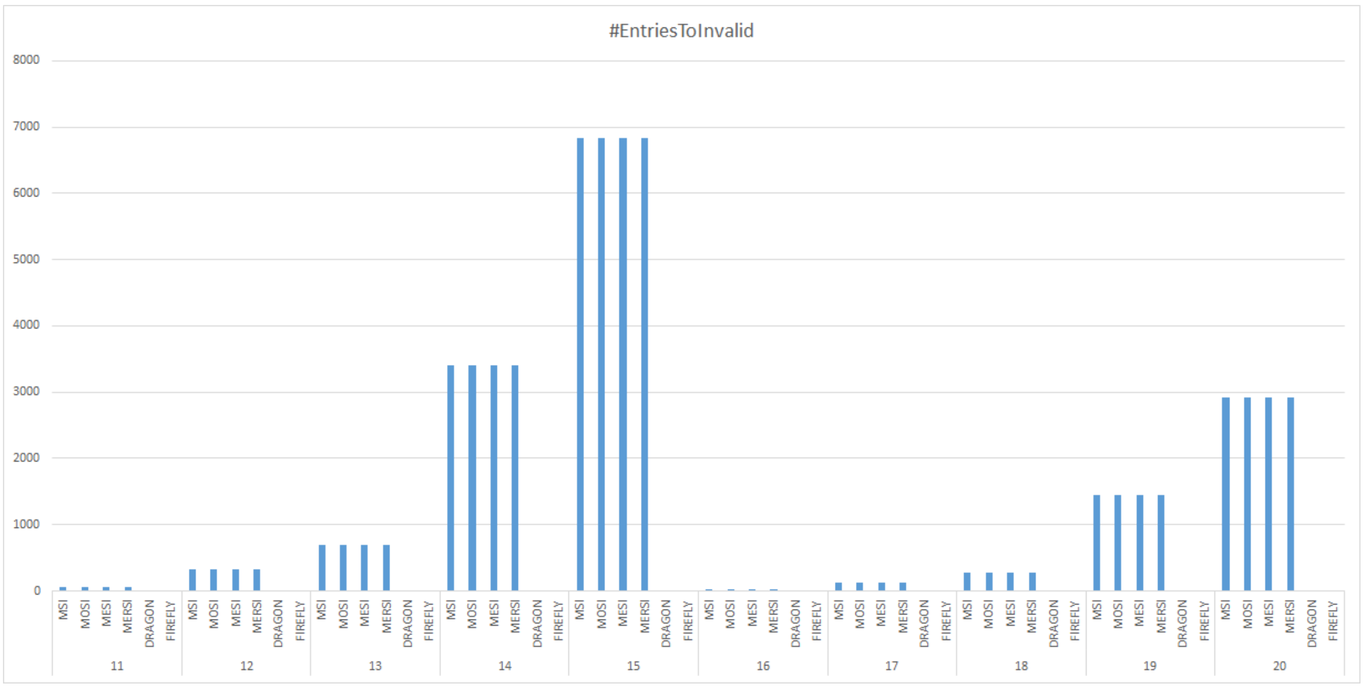


Figure 34. Entries to invalid Scenario 11-20

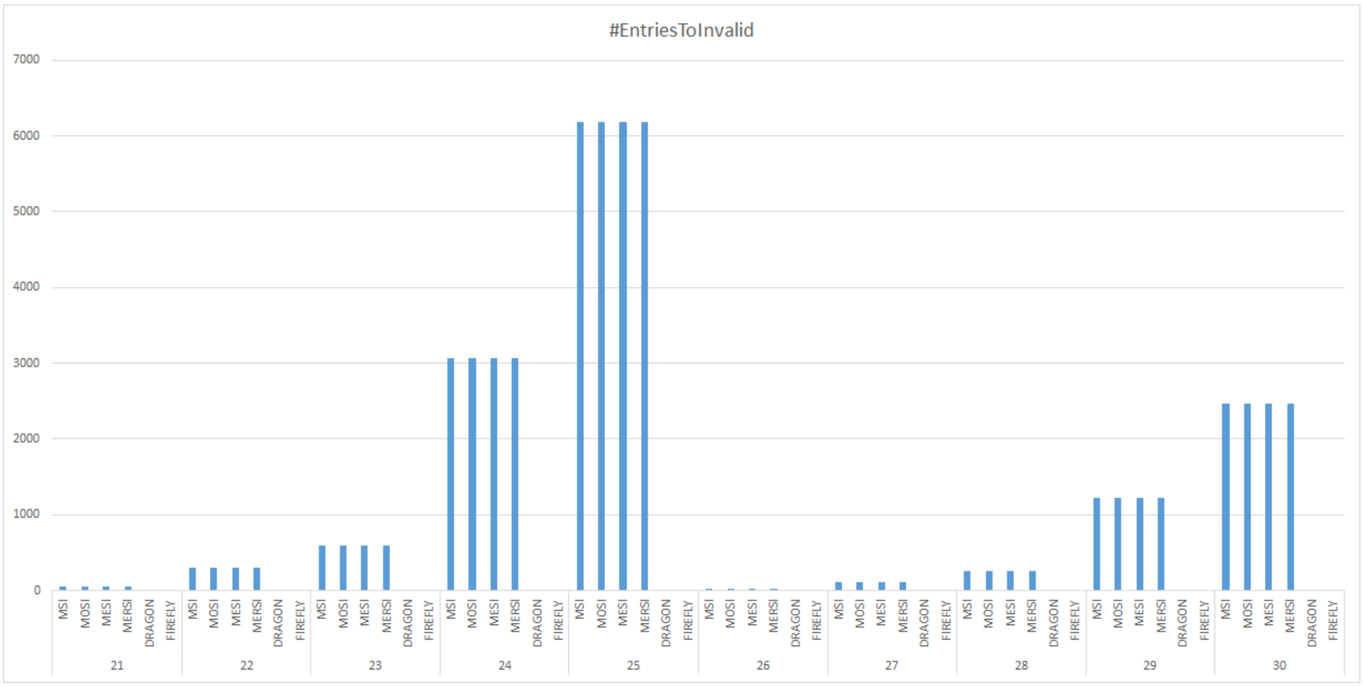


Figure 35. Entries to invalid Scenario 21-30

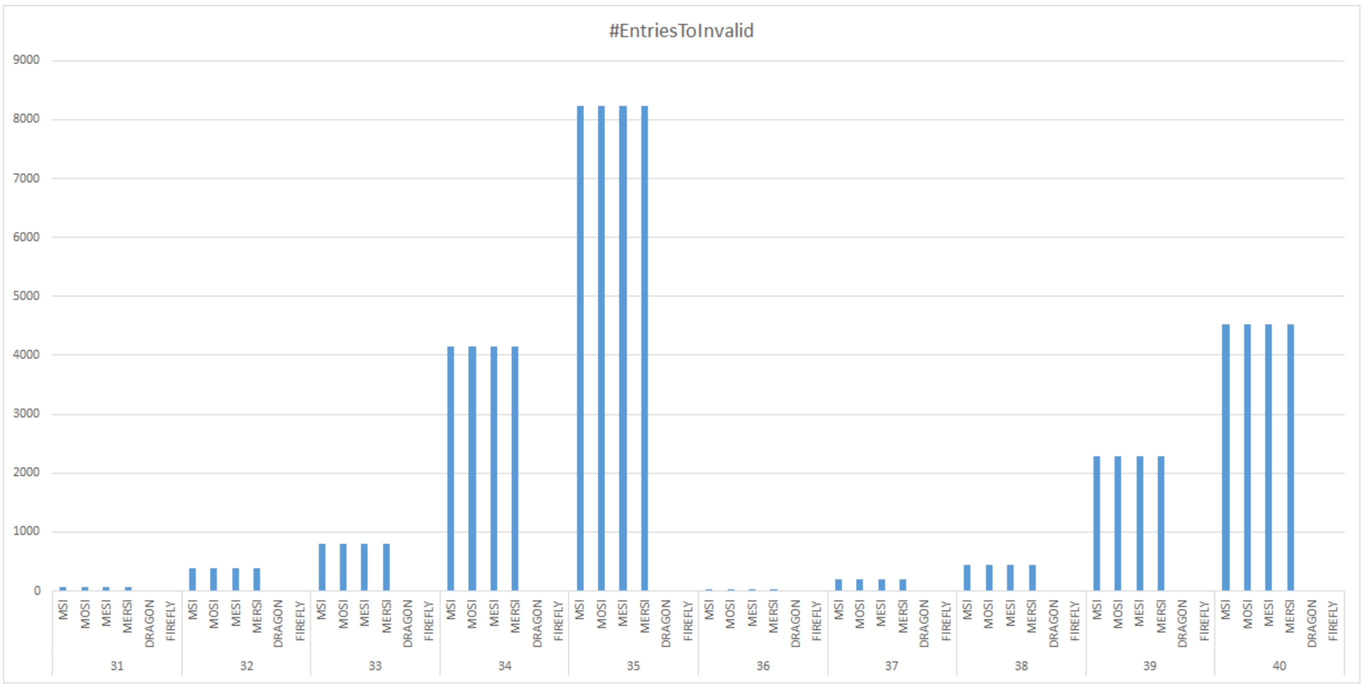


Figure 36. Entries to invalid Scenario 31-40

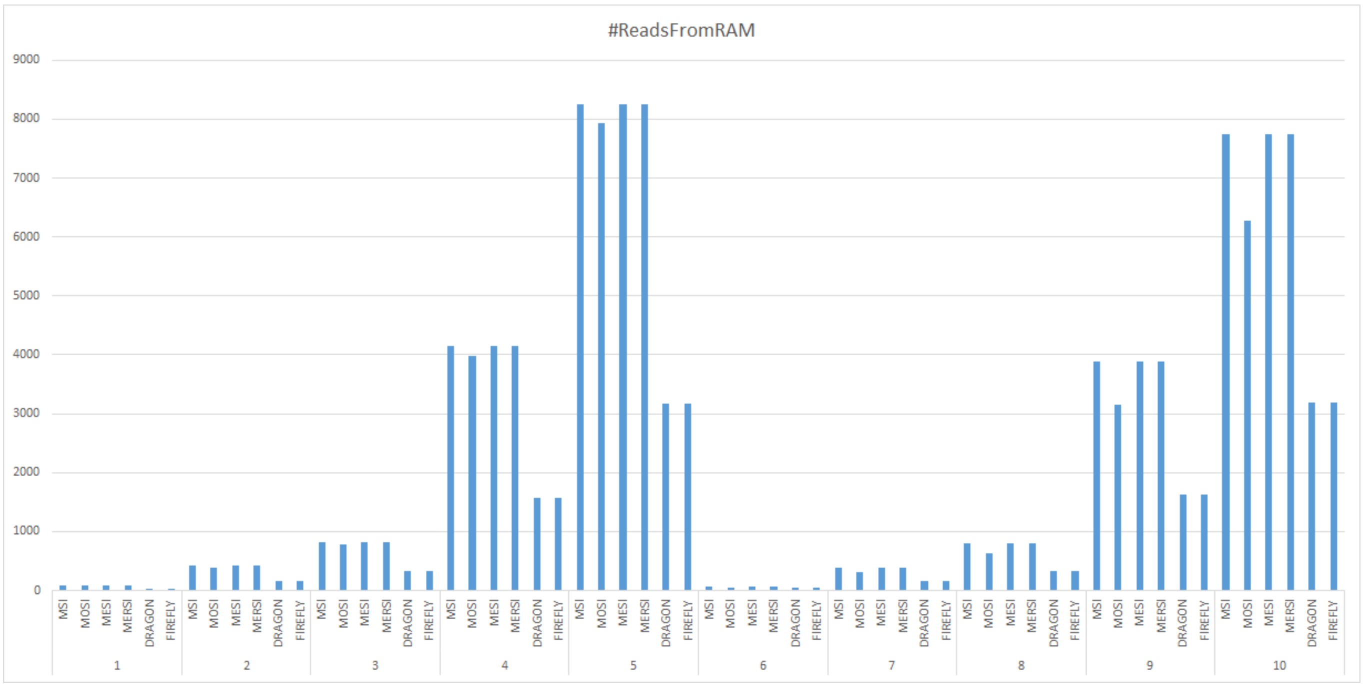


Figure 37. Reads from RAM Scenario 1-10

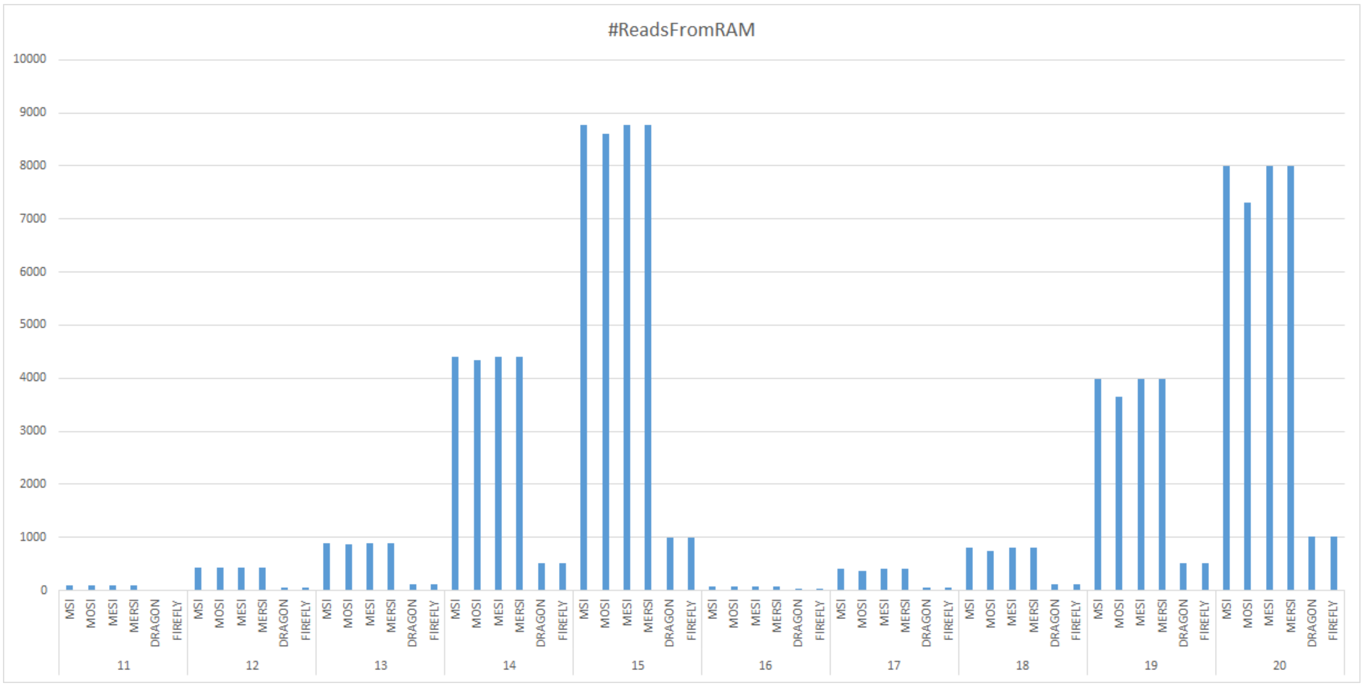


Figure 38. Reads from RAM Scenario 11-20

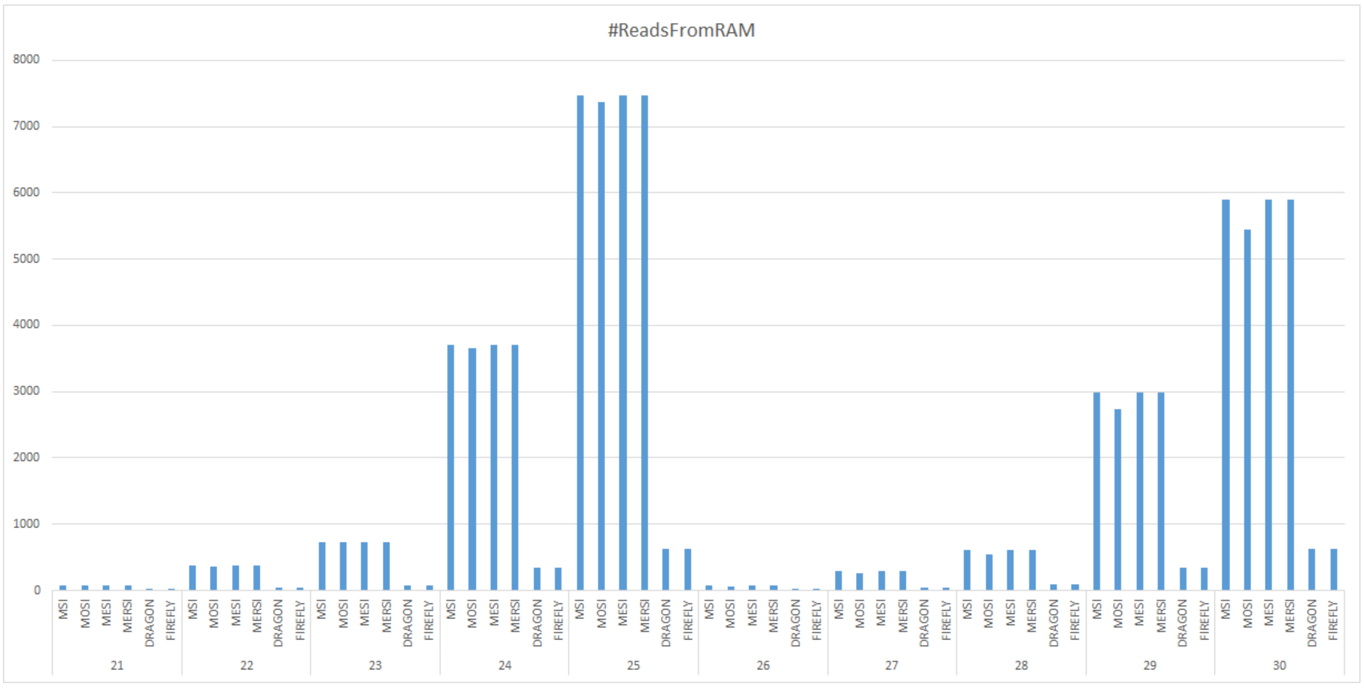


Figure 39. Reads from RAM Scenario 21-30

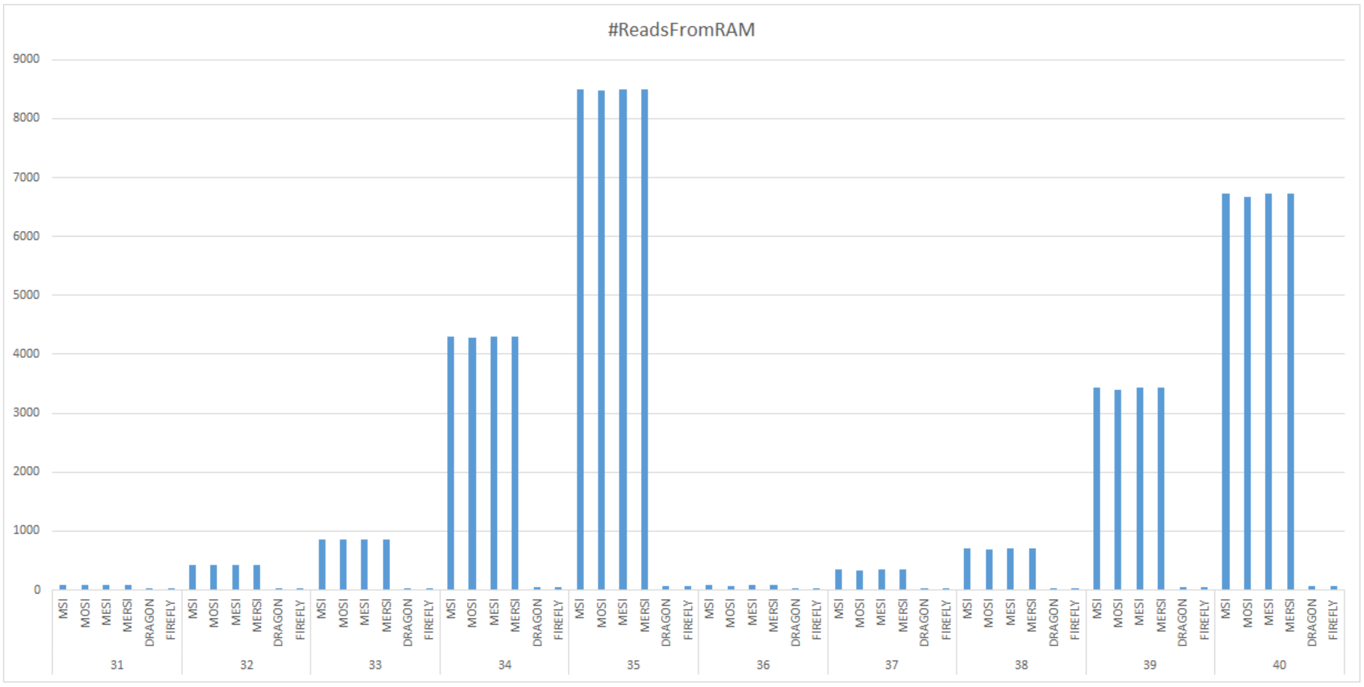


Figure 40. Reads from RAM Scenario 31-40

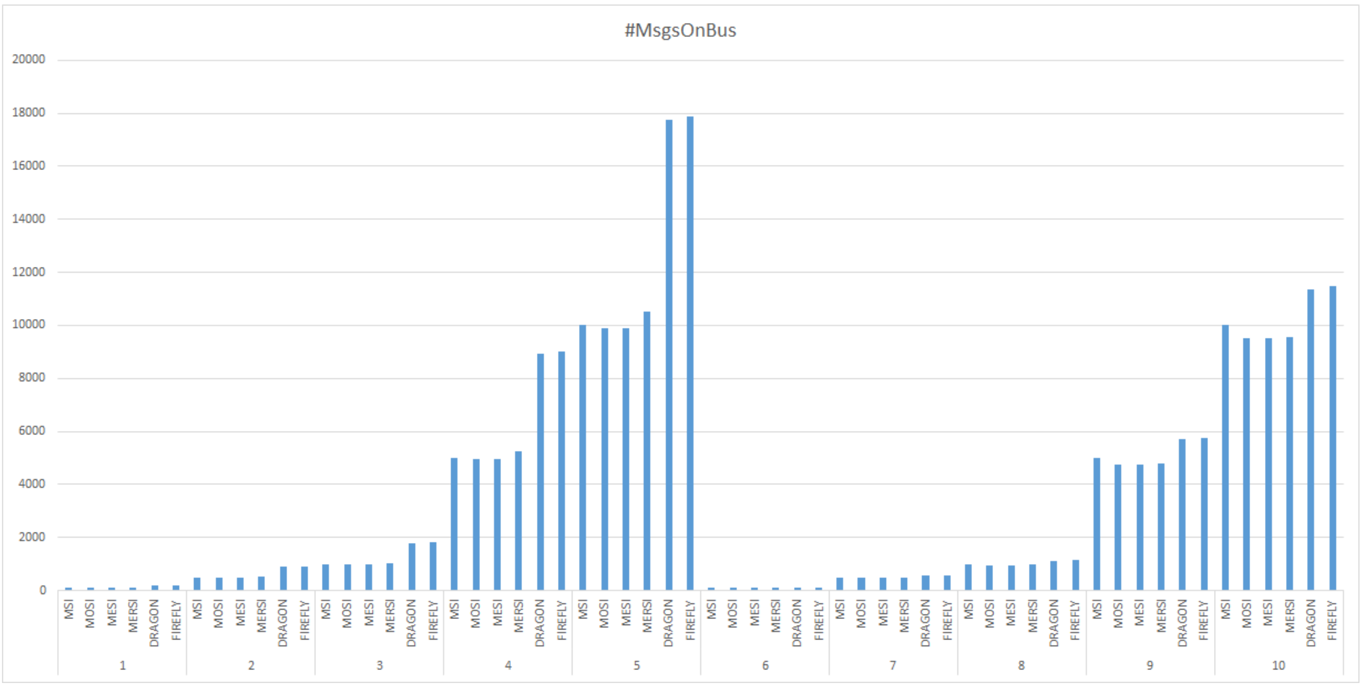


Figure 41. Messages on Bus Scenario 1-10

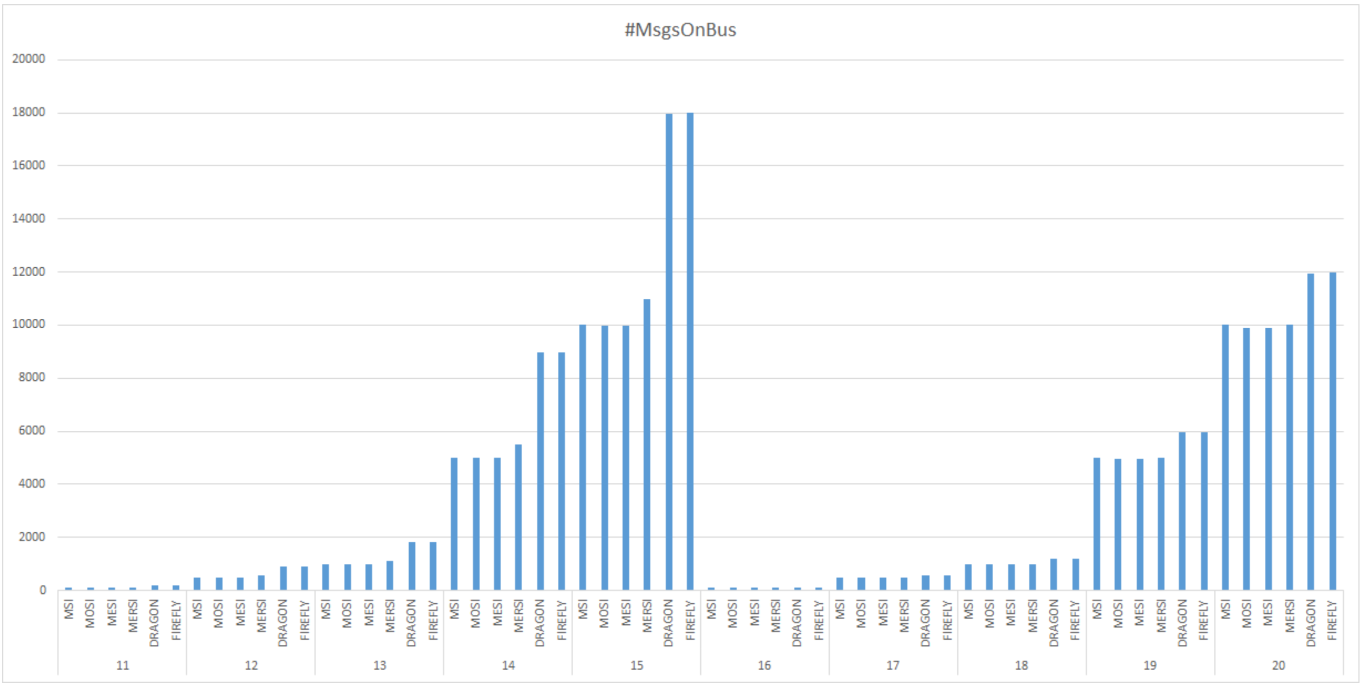


Figure 42. Messages on Bus Scenario 11-20

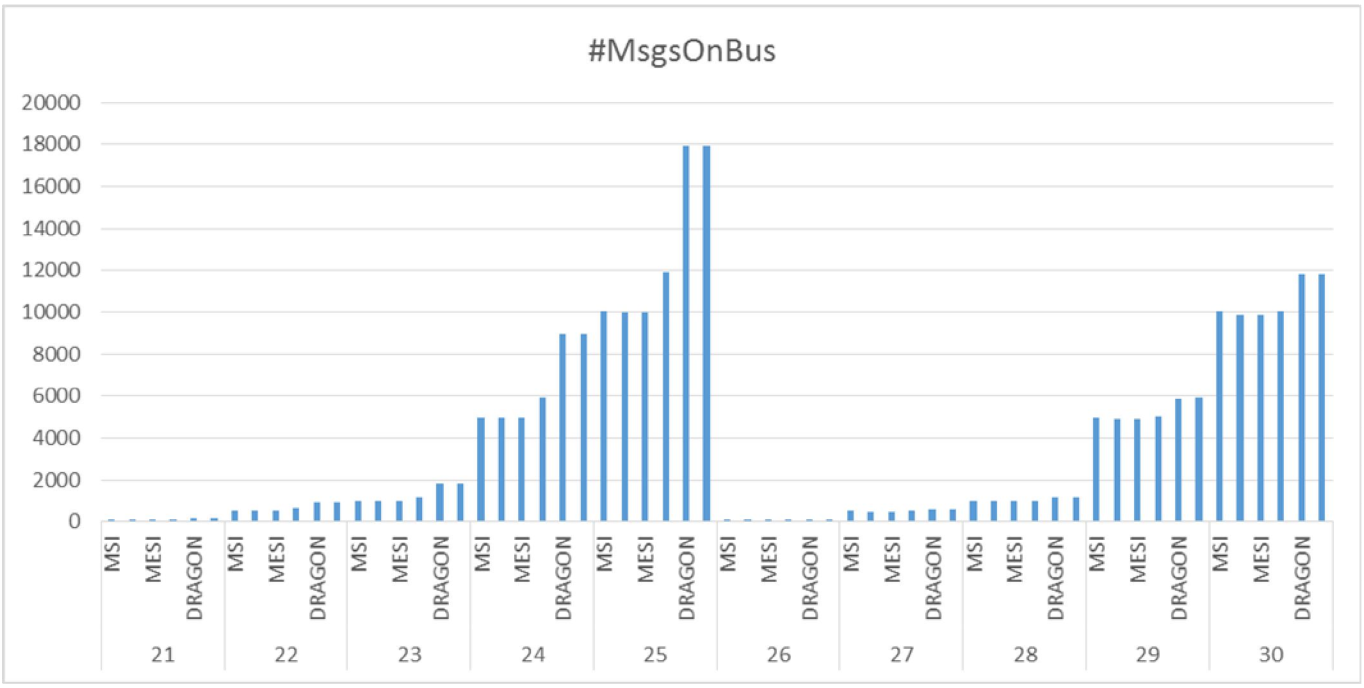


Figure 43. Messages on Bus Scenario 21-30

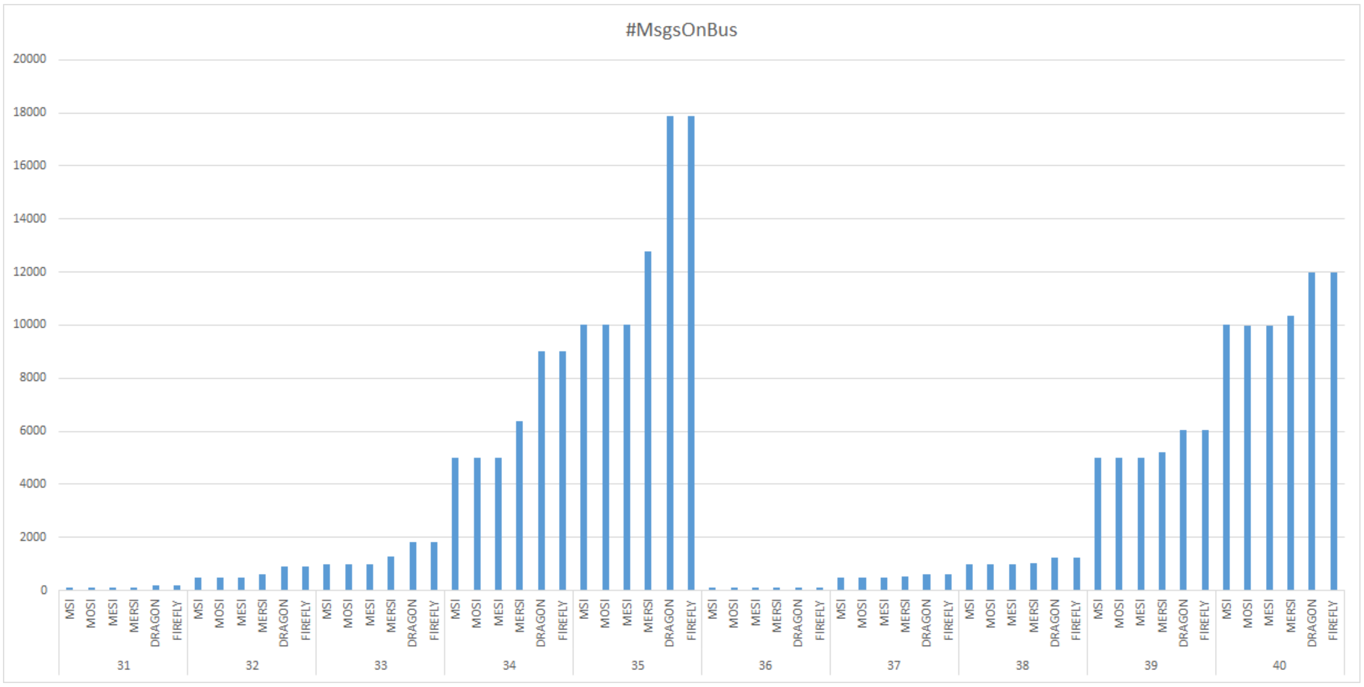


Figure 44. Messages on Bus Scenario 31-40

V. Result Analysis – Key Observations

1) Cache Misses

a) Cache coherence protocols do not have an impact on how many cache misses occur. Number of cache misses remains consistent across all protocols.

b) When cache holds more number of blocks, less number of cache misses occur.

2) Coherence Misses

a) MSI, MESI, MOSI and MERSI have the same number of coherence misses. These extensions are not aimed at improving the coherence misses, but they are aimed at improving other parameters that we discuss later.

b) Larger cache size results in more number of coherence misses. This is because there are more blocks in memory and the probability of them going into invalid state are greater.

c) DRAGON and FIREFLY have no Invalid State and thus technically (at least in the sense of this simulator) have no coherence misses.

3) Writebacks

a) MSI, MESI, MOSI and MERSI have the same number of writebacks happening as they all have invalid states and every remote read or remote write causes the processors to write the local block back to memory (given that they themselves have already modified it).

b) DRAGON and FIREFLY have very few writebacks taking place because the only time a writeback takes place is when the block is being swapped out of the local cache of a processor who knows that they have modified this block at some point.

4) Entries to Invalid

a) The number of entries to invalid state are consistent with the results seen in Coherence misses.

b) The number of entries to invalid state remain constant across MSI, MOSI, MESI and MERSI because in case of all these protocols, a block goes to invalid state every time a remote write is executed.

c) DRAGON and FIREFLY have no invalid state and thus for these protocols the number remains 0.

5) Reads from RAM

a) MOSI has less reads from RAM happening than MSI, MESI and MERSI because whenever a block is needed, a processor with that block in OWNED state can share it directly with the processor requesting the block. This means that the other processors do not always have to read from lower level memory in case of a cache miss.

b) DRAGON and FIREFLY have significantly less reads taking place from lower level memory because every time some write takes place, the updated data is propagated to all the processors. This way, whenever these processors want to read the most up to date data, they only have to go as far as their local cache and no further.

6) Messages on Bus

a) MESI and MOSI have slightly less messages being sent on the bus than MSI and MERSI because whenever a block is in E or O state, the processor does not need to broadcast a message on the bus saying that I want to read this block.

b) DRAGON and FIREFLY have almost double the messages being sent on the bus as every time a write takes place, the executing processor broadcasts that they are writing to this block as well as sending the new data that they plan on writing to all other processors. In this simulator, we consider this data as just one more message, but in real life the size of data will have a significantly higher impact.

VI. CONCLUSION

We have implemented a multiprocessor cache coherence simulator and evaluated 6 snooping based protocols as a part of our project. Considering the empirical evaluation, we can see that there are certain circumstances under which each of the protocols should be used. MOSI reduces the number of reads from RAM and thus saves the delay added due to the same. MOSI and MESI reduce the number of messages that are sent on the bus. But at the same time, Dragon and Firefly have significantly less number of reads and number of writebacks. This gives them a great advantage, but looking at the number of messages exchanged by processors under these protocols, it is clear that they hinder the scalability of the architecture. There are several other trade-offs that need to be considered while choosing an approach for coherence enforcement and this report along with the simulator gives a user the capability to evaluate these approaches.

VII. References

[01] "What Is Cache Coherence? Webopedia Definition". Webopedia.com. N.p., 2016. Web. 29 Mar. 2016.

[02] "Cache Coherence". Wikipedia. N.p., 2016. Web. 29 Mar. 2016.

[03] James Archibald and Jean-Loup Baer. 1986. Cache coherence protocols: evaluation using a multiprocessor simulation model. ACM Trans. Comput. Syst. 4, 4 (September 1986), 273-298.

[04] "What Is Snooping Protocol? - A Definition from The Webopedia IT Dictionary". Webopedia.com. N.p., 2016. Web. 29 Mar. 2016.

[05] "Cache Coherence Problem - Georgia Tech - HPCA: Part 5". YouTube. N.p., 2016. Web. 29 Mar. 2016.

[06] "Cache Coherence". Slideshare.net. N.p., 2013. Web. 29 Mar. 2016.

[07] Lawrence, Ramon. "A survey of cache coherence mechanisms in shared memory multiprocessors." Proceedings of the International Conference on Advances in Computer Science and Electronics Engineering. 1998

[08] A. Agarwal, R. Simoni, J. Hennessy and M. Horowitz, "An evaluation of directory schemes for cache coherence," Computer Architecture, 1988. Conference Proceedings. 15th Annual International Symposium on, Honolulu, HI, 1988, pp. 280-289.

[09] Daniel J. S. Mark D. H. David A. W., "A Primer on Memory Consistency and Cache Coherence," Morgan Claypool Publishers, 2011.

[10] "MESI, MOSI, MOESI Solution Quiz - Georgia Tech - HPCA: Part 5". YouTube. N.p., 2016. Web. 29 Mar. 2016.

[11] "19 5 L18S5 Cache Coherence Protocols". YouTube. N.p., 2016. Web. 29 Mar. 2016.

[12] S, M., & K, D. (2013). Hybrid Cache Coherence Protocol for Multi-Core Processor Architecture. International Journal of Computer Applications, 70(14), 24-29. doi:10.5120/12031-8060

[13] S. Al-Hothali, S. Soomro, K. Tanvir, R. Tuli, "Snoopy and Directory Based Cache Coherence Protocols: A Critical Analysis," Information & Communication Technology, vol.4, no.1, pp.1,10, 2010.

[14] J. Li et al., "A New Kind of Hybrid Cache Coherence Protocol for Multiprocessor with D-Cache," Future Computer Science and Education (ICFCSE), 2011 International Conference on, Xi'an, 2011, pp. 641-645.

[15] J. Li, L. Shi, Q. Li, C. J. Xue and Y. Xu, "Thread Progress Aware Coherence Adaption for Hybrid Cache Coherence Protocols," in IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 10, pp. 2697-2707, Oct. 2014.

APPENDIX A – CODE SNIPPETS

1. Protocol Interface

```
public interface ProtocolInterface{
   public int numberOfStates=0;
   public void localRead(int procID,int blockID,boolean isShared);
   public void localWrite(int procID,int blockID, boolean isShared);
   public void remoteRead(int procID, int blockID);
   public void remoteWrite(int procID, int blockID, double value);
}
```

2. User Choices Class

```
public class UserChoices {
   public static int numberOfProcessors = 2;
   public static int numberOfBlocksLocal = 2;
   public static int numberOfBlocksMain = 2;
   public static double wbDelay = 0;
   public static ProtocolEnum protocol = ProtocolEnum.MSI;
   public static String inputSequence = "";
   public static ProtocolInterface useProtocol;

   public static void updateProtocolObject() {
      switch (protocol) {
         case MSI:
            useProtocol = new MSI();
            break;
         case MOSI:
            useProtocol = new MOSI();
            break;
         case MESI:
            useProtocol = new MESI();
            break;
         case MERSI:
            useProtocol = new MERSI();
            break;
         case DRAGON:
            useProtocol = new Dragon();
            break;
         case FIREFLY:
            useProtocol = new Firefly();
            break;
      }
   }
}
```

3. Evaluation parameters

```
public class EvalParam {
   public int cacheReadMisses;
   public int cacheWriteMisses;
   public int coherenceReadMisses;
   public int coherenceWriteMisses;
   public int entriesToInvalid;
   public int cntMessagesOnBus;
   public int cntReadsFromRam;
   public int cntWriteBacks;
   }
```

4. Random Sequence Generator

```
private void jBtnGenRandomActionPerformed(java.awt.event.ActionEvent evt) {
      double writeThreshold = 0.5;
      if (jCmbDominant.getSelectedIndex() == 0) {
         writeThreshold = 0.8;
      } else if (jCmbDominant.getSelectedIndex() == 1) {
         writeThreshold = 0.2;
      } else {

      }
      int proc = Integer.parseInt(jTxtNumProc.getText());
      int ramBlocks = Integer.parseInt(jTxtNumBlockMainMemory.getText());
      String seq = "";
      for (int i = 0; i < Integer.parseInt(jTxtNumOps.getText()); i++) {

         int selctedProc = SimulatorWindow.selectBasedOnRandom(proc, Math.random());
         int selectedBlock = SimulatorWindow.selectBasedOnRandom(ramBlocks, Math.random());

         seq += "P-" + selctedProc + ":B-" + selectedBlock + ":";
         if (Math.random() > writeThreshold) {
            seq += "R";
         } else {
            seq += "W:" + Math.round(Math.random() * 100);
         }
         if (i < Integer.parseInt(jTxtNumOps.getText()) - 1) {
            seq += ",";
         }
      }
      SimulatorWindow.jTxtScenario.setText(seq);

   }
```

5. Sequence Executor

```
public static void executeScenario(String scenario) throws InterruptedException {
        SimulatorWindow.setControllerState(false);
        processors = new Processor[UserChoices.numberOfProcessors];
        for (int i = 0; i < UserChoices.numberOfProcessors; i++) {
            processors[i] = new Processor(UserChoices.numberOfBlocksLocal);
        }
        ram = new SharedMemory(UserChoices.numberOfBlocksMain);
        UserChoices.updateProtocolObject();
        String[] ops = scenario.split(",");
        for (int i = 0; i < ops.length; i++) {
            String[] spec = ops[i].split(":");
            int procId = Exec.getID(spec[0]);
            int blockId = Exec.getID(spec[1]);
            for (int k = 0; k < processors[procId].localCache.length; k++) {
                if (processors[procId].localCache[k].blockID != blockId) {
                    processors[procId].localCache[k].lastUsed++;
                }
            }
            String op = spec[2];
            double val = 0;
            if (op.equals("W")) {
                val = Double.parseDouble(spec[3]);
            }
            if (op.equals("R")) {
                Bus.readBlock(processors, procId, blockId);
            } else {
                Bus.writeBlock(processors, procId, blockId, val);
            }
            SimulatorWindow.updateSequenceExecutionStatus();
            Thread.sleep(500);
        }
        for(int x=0;x<processors.length;x++){
            for(int z=0;z<processors[x].localCache.length;z++){
                if(processors[x].localCache[z].blockState==StateEnum.D ||
                        processors[x].localCache[z].blockState==StateEnum.M ||
                        processors[x].localCache[z].blockState==StateEnum.SD){
                    ram.values[processors[x].localCache[z].blockID]=processors[x].localCache[z].value;
                    SimulatorWindow.evaluator.cntWriteBacks++;
                }

            }
        }
        SimulatorWindow.setControllerState(true);
        SimulatorWindow.updateEvalParamBox();
    }
```