Multi-Processor Cache Coherence Simulator

Oluwaseyi Bello
Carnegie Mellon University
Pittsburgh, USA
obello@andrew.cmu.edu

Darshil Kaneria
Carnegie Mellon University
Pittsburgh, USA
dkaneria@andrew.cmu.edu

Abstract

Multi-core caches present unique challenges that are absent in single-core caches. When multiple caches have the same cache lines replicated within them, different processors can observe different values for the same memory location. To solve this problem, cache coherence protocols are used to ensure that different caches have a consistent state for the same cache line that is replicated across them. This project aims to explore two major types of cache coherence protocols by creating a simulator to simulate them and running various workloads. The focus is on how different coherence protocols perform under different workloads, and conclusions will be drawn from our analysis.

1 Introduction

Due to the growing gap between CPU speed and memory access times, caches are used to improve memory access latency, with various levels (L1, L2, L3) of caches. The access times and capacity of the caches increase from left to right (L1 < L2 < L3). Together, the entire cache hierarchy significantly reduces memory access times.

In a uni-processor system, there is a single L1[1], L2, and L3 cache. However, in a multi-processor system, there are multiple L1 and L2 caches with a single L3 cache[2]. There is a single L1 and L2 cache per core, and all cores share a single L3 cache, also known as the last-level cache. The presence of multiple L1 and L2 caches means that a cache line can exist in different caches at the same time. This presents unique challenges in keeping the value of that cache line consistent across all caches. Specifically, the cache line can have multiple values in different caches, implying that multiple cores can observe different values for the same memory location. This breaks the programmer's expectation of how memory should work, and to solve this problem, two major types of coherence protocols have been developed. They are:

(1) Snooping-Based Coherence Protocols.
(2) Directory-Based Coherence Protocols.

Snooping-based protocols work by broadcasting read/write requests on the bus, so that all caches can see them and take action based on the specific snooping-based protocol to ensure a cache line is consistent across individual caches.

Directory-based protocols work by having a central point known as the directory that receives all read/write requests from the caches, maintains the state of the line, and notifies the necessary caches that need to be updated. This avoids broadcast and uses point-to-point communication. The directory that stores the global state for a cache line ensures that a cache line remains consistent across individual caches.

[1]There are instruction and data caches
[2]There could be multiple L3 caches e.g. NUMA systems

The two types of protocols work very differently, and this project aims to explore these differences by using both synthetic and real-life workloads to understand their inner workings.

First, we will create a simulator to simulate the various types of coherence protocols, which we will use to run our analysis. Using the workloads, an extensive analysis will be conducted on the various protocols. At the end of the analysis, the limitations and observations of each protocol will be documented, and conclusions will be drawn on how they perform under different workloads.

2 Background

Cache coherence in shared memory systems is necessary to solve the specific problem where multiple cores are accessing a cache line simultaneously, and one of those accesses is a write. The danger is reading stale data (another processor has updated it, but the current processor has not yet seen the update) or reading into the future (the current processor has made a change to the line and is reading it again, but other processors have not seen the change yet).

For example, consider the sequence of events

(1) P1 reads A.
(2) P2 reads A.
(3) P1 modifies A.
(4) P1 reads A.
(5) P2 reads A.

Here, P1 and P2 will observe different values, which breaks the programmer's expectation. Thus, it is necessary to keep the values of A consistent in both caches, or more specifically, to keep the cache line that contains A consistent in both caches.

How to keep this consistent includes ensuring the following:

(1) **Program Order**
If a processor P1 writes to an address A, and then reads from that same address, if no other processor wrote to A between the write and the read, the value written initially should be returned. This is the same expectation in a uni-processor system.
(2) **Write Propagation**
When a processor P1 writes to an address A, if another processor P2 reads that same address, it should get the value that P1 wrote, provided that the time between the write and the read is sufficiently separated in time and assuming no other write occurs between P1's write and P2's read. This ensures that a processor holding a cache line is aware of writes made by another processor to that same cache line.
(3) **Write Serialization** In the case of multiple writes to the same address A, all processors should observe the writes in a consistent order; the ordering here is not defined, but all processors' views of the writes should be consistent. This

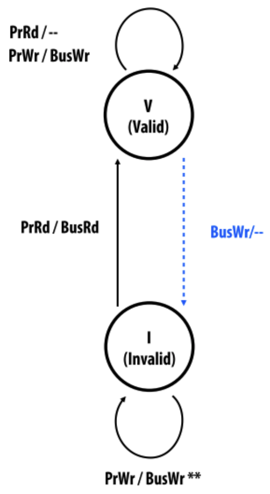

Figure 1: MI state machine ([1])

ensures that all processors see all the changes to a specific cache line in the same order.

To achieve this, two main types of coherence protocols have been developed, and we will explore them further.

2.1 Snooping-Based Protocols

Snooping-based protocols operate on the notion of alerting every cache controller to any operation (read or write) that is occurring. Since every cache controller is aware of every operation that occurs, they can take individual actions locally based on a defined rule or protocol, and the result of these individual actions will ensure coherence.

A key point in snooping is that the interconnect has to support broadcasts as reads and writes from a processor are broadcast to all other caches, and that the bus enforces a total order on how processors see the messages[3]

The protocols that the cache controllers use to change their local state are described below.

(1) **MI**
There are two states a line can be in: Valid (M) or Invalid (I).
M - Only the cache has access to the line and can read/write to it. The line might be dirty or clean.
I - line is invalid in the cache.
Every read or write to a cache line will cause an invalidation in any other caches that have that line in the Valid (M) state.

(2) **MSI**
The issue with MI is that reads cause invalidations all the time. MSI improves this by adding a Shared state(S).
S - Multiple caches have access to the line, and they can only read, not write. To write, they will have to move to M and invalidate the state in other caches. The line is clean.
M - Now represents a line that is dirty. Only the cache still has access to the line.

[3]There are advanced designs that do not rely on a total order

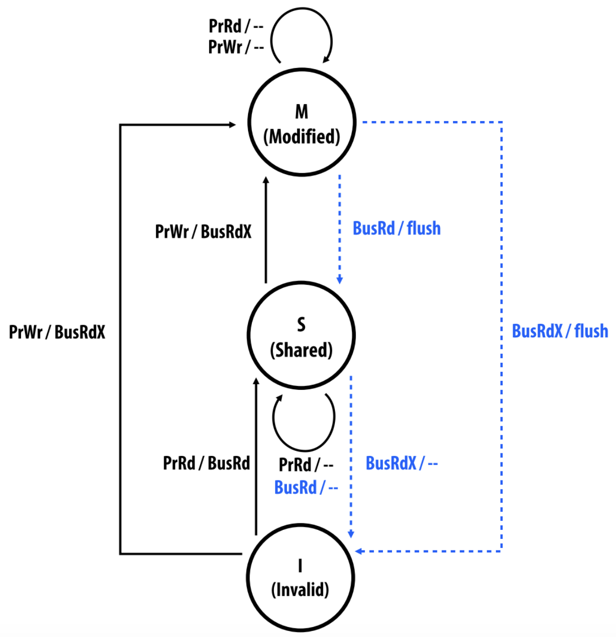

Figure 2: MSI state machine ([1])

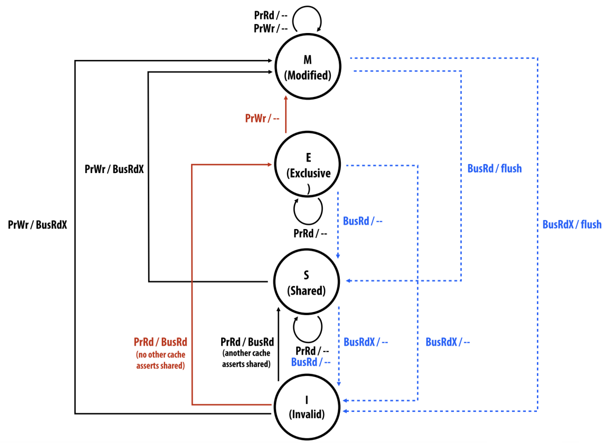

Figure 3: MESI state machine ([1])

(3) **MESI**
The issue with MSI is that if a write to the same line follows a read, the cache will first request the line in Shared (S) state and then have to request it again in Modified (M) state, generating extra traffic and adding latency. MESI improves on MSI by adding an Exclusive state (E).
E - Only the cache has access to the line, and the line is clean. It can read the line, and if it needs to write to the line, it can silently upgrade to M without notifying any other caches.
Now, when a read is requested for a line, it is first put in Exclusive (E) state, instead of Shared (S) state. When a write

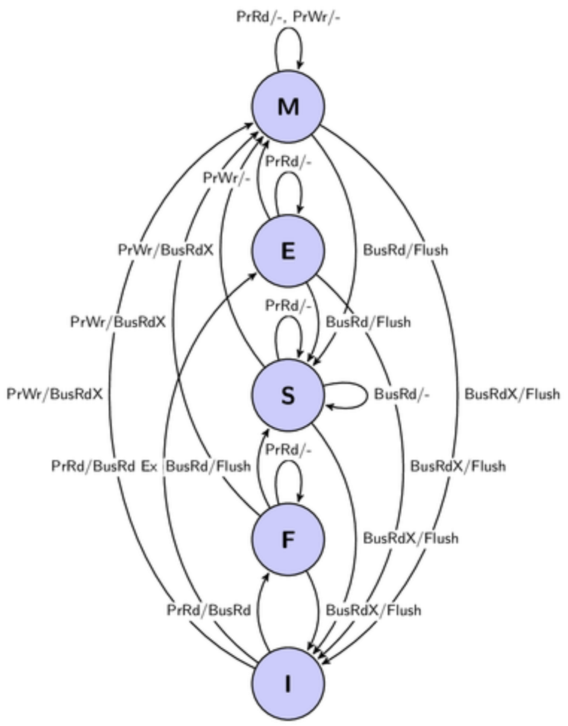

Figure 4: MESIF state machine ([4])

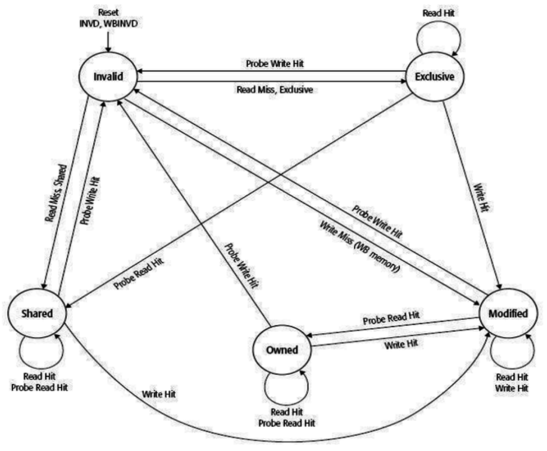

Figure 5: MOESI state machine ([3])

is then requested, it is silently upgraded to the Modified (M) state without having to notify other caches. This helps to cut down on interconnect traffic as well as the latency. Downgrades from Exclusive (E) to Shared (S) can also occur if another cache wants to read the line. Downgrades from Exclusive(E) to Invalid(I) can also occur if another cache want to write to the line.

(4) **MESIF**
The limitation with MESI is that when a line is in shared mode in a group of caches, if another cache makes a request for the line, all the caches that have it in shared mode respond with the data for the line, causing significant traffic on the interconnect. MESIF improves on this by adding a Forwarder (F) state.
F - Similar to S, but in addition, the cache is responsible for providing the data to this line.
Now, when a request is made to the shared line, only the forwarder responds with data for that line. The last cache to make a read request for the line is the new Forwarder (F), and the cache that provided the data (previously the Forwarder(F)) becomes Shared (S). This significantly reduces interconnect traffic.

(5) **MOESI**
Similar to MESIF, MOESI improves on MESI by adding an Owned state(O) **O** - The cache owns the line, the line is dirty, and the cache is responsible for providing the data. Other caches can have the line in Shared (S) mode, but they do not provide data when a request is made for the line. Instead of downgrading from Modified (M) to Shared (S) when a read request is made for the line, the cache downgrades from Modified (M) to Owned (O) and doesn't flush to memory. Any additional sharer will be in the S state, but only the cache in the Owned (O) state will supply the data. Unlike in MESIF, the Owned (O) state does not change to the latest sharer when a read request is made. It remains with the cache that downgraded from Modified (M) to Owned (O).
Now, when a request is made for the shared line, only the Owner responds with data for that line, significantly reducing interconnect traffic.

2.2 Directory-Based Protocols

Directory-based protocols operate on the notion of having a centralized point that every cache controller sends requests for any operation (read or write) that is occurring. The directory can then notify any other cache controllers that require the information, and it also maintains the state of the line in all caches. The combination of being the central point, being aware of all operations, and maintaining the state of the line in all caches allows the directory to ensure coherence. Every cache line is associated with a directory, also known as its home node.

A key point in the directory-based scheme is that the interconnect does not have to support broadcasts, as all messages are unicast between cache controllers and the directory, or between cache controllers themselves.

The directory tracks details about cache lines, such as which processors have the line in their caches and whether the line is dirty in any processor. It updates this information dynamically based on requests or replies from cache controllers. It can also send messages to caches for them to change their local states.

The directory can become a bottleneck, and to alleviate this, distributed directories can be used. In a distributed directory, different cache lines have different home nodes. This is used to scale

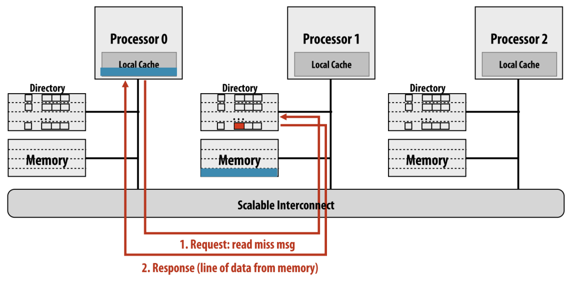

Figure 6: Read Miss to Clean Line ([2])

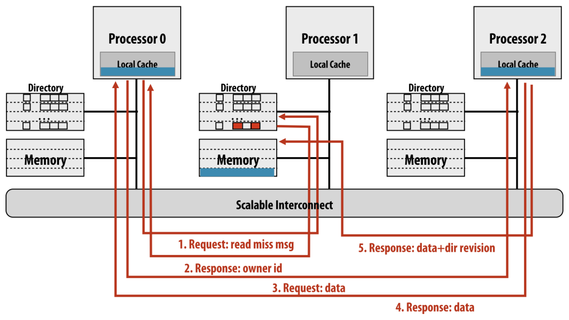

Figure 7: Read Miss to Dirty Line ([2])

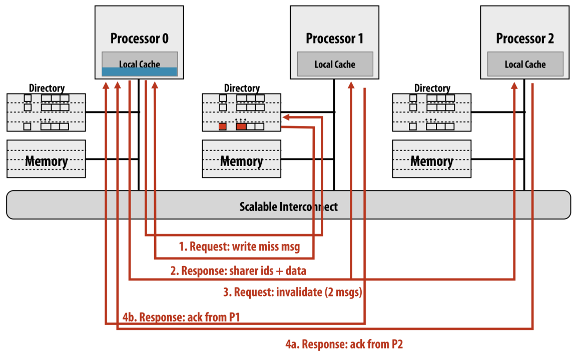
Processor 0
Local Cache
Processor 1
Local Cache
Processor 2
Local Cache
Directory
Directory
Directory
Memory
Memory
Memory
Scalable Interconnect
1. Request: write miss msg
2. Response: sharer ids + data
3. Request: invalidate (2 msgs)
4b. Response: ack from P1
4a. Response: ack from P2

Figure 8: Write Miss to Clean Shared Line ([2])

to a large number of processors in NUMA (Non-Uniform Memory Access) systems.

The directory scheme can be combined with any of the snooping protocols described earlier, as each cache still maintains a local state for any line it holds. The main difference in the directory scheme is the message communication structure that changes from broadcast to unicast. Below, we describe the flow of some directory messages.

- **Read Miss To Clean Line**
  Since the line is clean, this means that memory can provide the data. The read request is sent to the home directory for the line, and it responds with the data for that line. The directory also updates its state to indicate that a cache has the line in shared state.

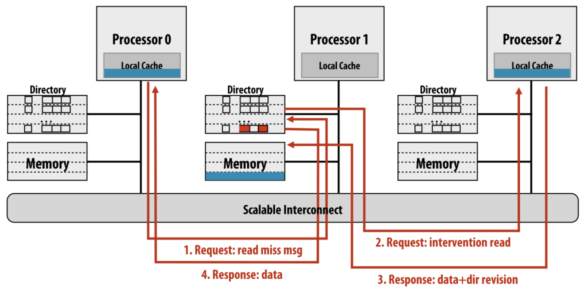

Figure 9: Intervention Forwarding ([2])

- **Write Miss To Clean Line**
  Since the line is clean, this means that memory can provide the data. The write request goes to the home directory for the line, and it responds with the data for the line. The directory also updates its state to indicate that a cache has the line in dirty state and that the cache is now the owner of that line.
- **Read Miss to Dirty Line** For a dirty line, since some cache is responsible for providing the data, after the read request is sent to the home directory, the directory responds to the requesting cache with the owner's information, and the requesting cache then contacts the owner to fetch the line. The owner responds with the line and also notifies the directory so that it can update its state to indicate that multiple caches are sharing that line.
- **Write Miss to Dirty Line** For a dirty line, since some cache is responsible for providing the data, after the write request is sent to the home directory, the directory responds to the requesting cache with the owner's information, and the requesting cache then contacts the owner to fetch the line. The owner responds with the line, invalidates its local state, since there is a new owner, and also notifies the directory so that it can update its state to indicate the new owner of that line.
- **Read Miss to Clean Shared Line** For a shared line, since one or more caches have the line locally, after the read request is sent to the home directory, the directory responds to the requesting cache with the cache responsible for providing the line. The requesting cache then contacts the cache responsible for providing the line to fetch the line. The cache responsible for the line responds with the line. The directory also updates its state to indicate an additional sharer on the line.
- **Write Miss to Clean Shared Line** Since one or more caches have the line locally, after the write request is sent to the home directory, the directory responds with the sharer IDs and the data. The requesting cache then requests the sharers to invalidate the line. The sharers respond to the requesting line after invalidating. The directory also updates its state to indicate that there is now an owner of that line.

To reduce the number of messages sent, there is an optimization called intervention forwarding for the directory. This allows the

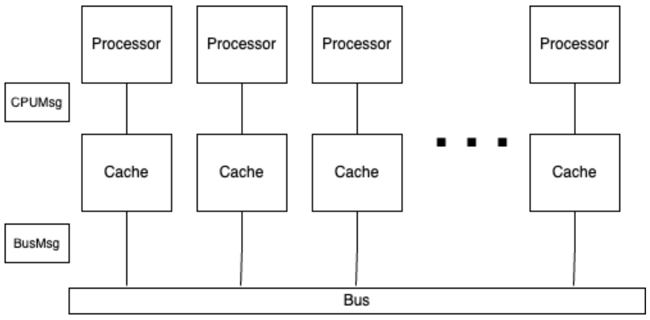

Figure 10: Snooping Design

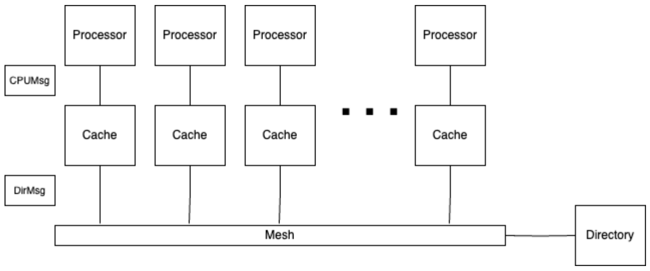

Figure 11: Directory Design

directory to request the line from the owner and forward it back to the requester, rather than the directory responding with the owner's information and the requester sending another message to the owner to request the line. This reduces the number of messages sent on the interconnect.

3 Experimental Methodology

To simulate the behavior of cache coherence protocols, we needed a Computer Architecture simulator that could model the handling of memory requests. CADSS [5] is an open-source computer architecture simulator that supports simulations in various areas of computer architecture. While CADSS meets all our requirements, we decided to build our simulator to scope down the features to our specific requirements, and also as a good learning experience.

We drew ideas from CADSS to incorporate into our simulator. We focused on simulating the processor, cache, interconnect, and directory components, as these components fully support what we require for our simulation.

3.0.1 Snooping Architecture. The processors each connect to a cache, which connects to a bus interconnect that supports broadcast messages. The messages sent between the processors and caches are CPUMsgs, while the messages sent on the bus are BUSMsgs.

The bus is an atomic bus that supports one outstanding request at a time and arbitrates access between processors in a fair manner.

3.0.2 Directory Architecture. The processors each connect to a cache, which connects to a mesh interconnect that supports unicast messages. The messages sent between the processors and caches are CPUMsgs, while the messages sent on the mesh are DirMsgs.

The directory processes one outstanding request at a time.

3.0.3 Trace. A trace is a representation of a memory operation and is read from files supplied at the start of the program. A trace line is represented as an Instruction.

```
 enum class OperationType
    {
        MEM_LOAD,
        MEM_STORE
    };
struct Instruction
    {
        OperationType command;
        size_t address;
    };
```

An Instruction has an OperationType, which can be a load or store, and an address.

3.0.4 Trace Reader. During a cycle, if the processor can accept requests, the trace reader reads a trace line from the trace files, converts it into an instruction, and passes the instruction to the processor.

3.0.5 Processor. The processor can have only one outstanding request to its cache. During a cycle, the processor checks if it can send a message to the cache, and if so, it sends a CPUMsg to the cache.

```
 enum class CPUMsgType
    {
        REQUEST,
        RESPONSE,
    };
 struct CPUMsg
    {
        CPUMsgType msgtype;
        Instruction inst_;
        size_t proc_;
        size_t proc_seq_;
    };
```

A CPUMsg contains a CPUMsgType, which can be a request or response, an Instruction, the processor that is sending the message, and a sequence number used to track responses from the cache.

3.0.6 Cache. The cache receives a request from the processor and checks if it is a hit. If it is a hit, it responds with a reply to the processor; otherwise, it begins a coherence operation.

Depending on the coherence type, it sends either a BusMsg or a DirMsg for that line, and when the coherence operation is done, it responds with a reply to the processor.

The different coherence states a line in the cache can be in are:

```
    enum class CoherenceState
    {
        MODIFIED,
        INVALID,
        SHARED,
        EXCLUSIVE,
        OWNED,
        FORWARDER
    };
```

These correspond to the various states of the cache coherence protocols.

The different coherence protocols supported are:

```
enum class CoherenceProtocol
{
    MI,
    MSI,
    MESI,
    MOESI,
    MESIF
};
```

These correspond to the various cache coherence protocols.

The coherence types supported are:

```
enum class CoherenceType
{
    SNOOP,
    DIRECTORY
};
```

3.0.7 *Bus.*

During a cycle, the bus receives a request from the cache and processes it. The processing depends on the type of request received. A Request is broadcast to caches, except for the cache from which it was received, while the response is sent to the specific cache that sent the initial request.

When it broadcasts a request to all caches, if no cache provides the data, the bus will simulate memory and provide the data.

The structure of a BusMsg is as follows:

```
const size_t BROADCAST = 1000;

enum class BusMsgType
{
    READ,
    WRITE,
    UPGRADE,
    DATA,
    SHARED,
    MEMDATA
};

struct BusMsg
{
    BusMsgType type_;
    CPUMsg cpureq_;
    size_t src_proc_;
    size_t dst_proc_;
};
```

A BusMsg has a BusMsgType, a CPUMsg, which refers to the CPUMsg that triggered the initial bus request, a source and destination processor for the message (a special address is used for broadcasts).

The BusMsgType can be:

- READ: A cache is trying to read the line.
- WRITE: A cache is trying to write to the line.
- UPGRADE: A cache is trying to upgrade from Shared to Modified.
- DATA: A cache has responded with the data. The cache that responded has flushed the data.
- SHARED: A cache has responded with the data, and this line can be shared across multiple caches.
- MEMDATA: Memory responded with the data.

3.0.8 *Directory.*

During a cycle, the directory receives a request from a cache and processes it. The processing depends on the type of request received.

The directory stores an entry for each line.

```
enum class DirState
{
    UNCACHED,
    SHARED,
    EXCLUSIVE
};

struct DirEntry
{
    DirState state;
    size_t address;
    std::optional<size_t> owner;
    std::unordered_set<size_t> sharers;
};
```

The State can be either

- UNCACHED: No cache has this line locally.
- SHARED: One/more caches are sharing this line.
- EXCLUSIVE: Only one cache has exclusive access to this line.

The address refers to the address of the line. Owner refers to the cache that has this line exclusively, while the Sharers is a list of caches sharing the line.

The structure of a directory message is as follows:

```
 const size_t DIRECTORY = 3000;

enum class DirMsgType
{
    READ,
    WRITE,
    UPGRADE,
    DATA,
    SHARED,
    INVALIDATE,
};

struct DirMsg
{
    DirMsgType type_;
    CPUMsg cpureq_;
    size_t src_proc_;
    size_t dst_proc_;
};
```

Similar to a bus, a DirMsg has a DirMsgType, a CPUMsg, which refers to the CPUMsg that triggered the initial bus request, a source and destination processor for the message (a special address is used for the directory).

The DirMsgType can be:

- READ: A cache is trying to read the line.
- WRITE: A cache is trying to write to the line.
- UPGRADE: A cache is trying to upgrade from Shared to Modified.
- DATA: A cache has responded with the data. The cache that responded has flushed the data.
- SHARED: A cache has responded with the data, and this line can be shared across multiple caches.
- INVALIDATE: The receiving cache should invalidate the line.

The simulator can be launched from the command line and supports the following arguments:

- **num_procs:** Number of processors
- **directory:** Directory to load traces from
- **cohertype:** Coherence type. SNOOP|DIRECTORY
- **coherproto:** Coherence protocol. MI|MSI|MESI|MESIF|MOESI
- **cache_line_size:** Cache line size. Default 64
- **cache_size:** Cache size. Default 8192
- **diropt:** whether to use intervention forwarding to optimize the directory traffic.

An Example:

```
./mpcsim --num_procs=4 --directory="traces/our_traces/
    false_sharing" --cohertype=SNOOP --coherproto=MESI --
    diropt=true
```

The input traces take the following form: L|S Address. For example:

```
L 104
S 6924
```

3.1 Benchmark Suite

The benchmark suite is a combination of synthetic and real-life traces.

For the synthetic traces, we used the following traces.

- **false_sharing:**
  In this workload, multiple processors write to different addresses that belong to the same cache line. This simulates a situation where the intention is to write to different addresses, but these addresses happen to fall on the same cache line.
- **producer_consumer:**
  In this workload, one processor acts as a producer, writing to an address or a collection of addresses, while all the other processors read from that address or collection of addresses.
- **multiple_writers:**
  In this workload, multiple processors read and write to the same address or a collection of addresses.
- **multiple_readers:**
  In this workload, multiple processors read from the same address or a collection of addresses.
- **no_sharing:**
  In this workload, there is no address overlap between multiple processors reading and writing.
- **random:** In this workload, there is no order at all. This is just a random workload.
- **partial_proc_use:** In this workload, not all processors in the system are in use. A portion of the processors read from or write to addresses, while the other processors do nothing. This simulates a situation where an application uses only one core in a multi-core system or where it uses only a subset of the cores.

For real-life traces, we selected a trace from the Parsec benchmark suite.

- **blackscholes** This computes prices for stock options based on the black scholes mathematical pricing model.

3.2 Protocols Evaluated

We evaluated the workloads across a combination of snooping and directory cache coherence protocols that were described in the background. Specifically, the protocols are as follows.

3.2.1 Snooping.

- MI
- MSI
- MESI
- MESIF
- MESIF

3.2.2 Directory.

- **No Directory Optimization:** Without intervention forwarding, as described in the background section.
- **Directory Optimization:** With intervention forwarding, as described in the background section.

These are implemented according to the state machines described in the background section, and their design follows the information in the System Architecture section.

3.3 Evaluation Metrics

We gathered the following metrics

```
struct CacheStats
{
    size_t hits;
    size_t misses;
    size_t evictions;
};

struct InterconnectStats
{
    size_t traffic;
    size_t cache_control_traffic;
    size_t cache_data_traffic;
    size_t mem_data_traffic;
};
```

- **CacheStats**
  - **hits:** Cache hits.
  - **misses:** Cache misses.
  - **evictions:** Cache evictions. (Coherence evictions only)
- **InterconnectStats**
  - **traffic:** Total traffic on the interconnect.

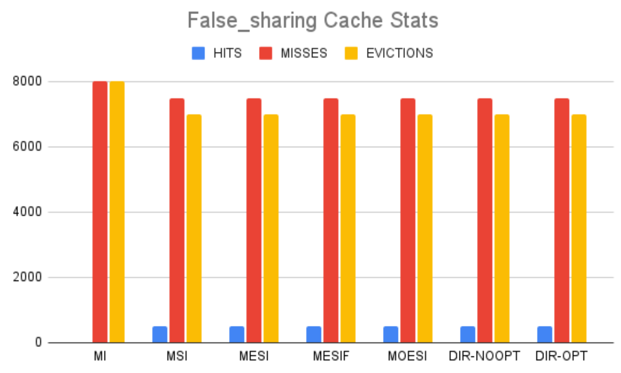

Figure 12: False Sharing: cache stats

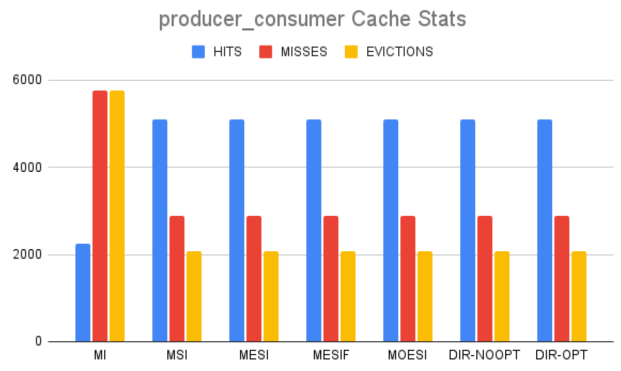

Figure 13: Producer Consumer: cache stats

- **cache_control_traffic:** Traffic due to coherence messages. These are coherence messages that do not involve the transmission of data.
- **cache_data_traffic:** Traffic due to caches supplying data. This means that a cache sent a request, and another cache is providing the data.
- **mem_data_traffic:** Traffic due to data supplied by the memory. This means that a cache previously sent a request, and since no cache provided the data, the memory is providing the data.

We ran the benchmark suite across the protocols to gather the evaluation metrics. We then analyzed the results and drew conclusions.

4 Performance Evaluation

4.1 Cache Traffic Analysis

We evaluated all workloads and analyzed cache traffic (hits, misses, and evictions). For the synthetic traces, we generated traces for eight processes. For the real-life trace, we used Black-Scholes with four processes.

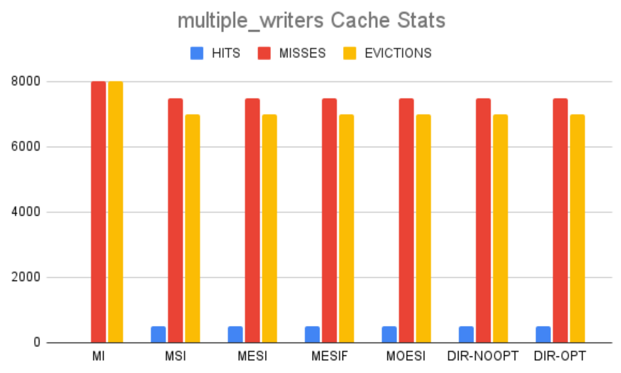

Figure 14: Multiple Writers: cache stats

4.1.1 ***False Sharing:*** MI showed a different pattern from the rest because MI cannot take advantage of sharing across multiple processors, since there is no notion of sharing in MI, all reads that are not hits cause evictions in the previous cache that also read the data, resulting in a higher number of evictions, which also results in a higher number of misses and fewer hits compared to the other protocols.

All others show a similar pattern. This is because, beyond MI, the optimizations from MSI to MESI to MESIF/MOESI are primarily about reducing interconnect traffic, rather than improving hits, misses, or evictions.

There is a high number of evictions because all accesses are to the same cache line, and each write causes invalidation in all other caches that were previously reading the data. The high number of evictions also corresponds to the high number of misses and the low number of hits.

4.1.2 ***Producer Consumer:*** Similar to False sharing, MI showed a different pattern from the others for the same reason, and all the others show a similar pattern.

There is a high number of hits since most processors read the data produced by another processor, and within a processor, there is some temporal and spatial locality in the reads. The evictions are caused by the writer writing to the cache line that the other processors previously read.

4.1.3 ***Multiple Writers:*** This traffic pattern closely follows the false sharing patterns, and the exact reasons for the noticed patterns are the same as those for false sharing.

4.1.4 ***Multiple Readers:*** The difference in traffic patterns between MI and others is similar to previous ones for the same reason. The differences here are more pronounced because there is a high amount of sharing between the processors.

The hit rate for the other protocols is very high because there is a high amount of temporal and spatial locality within a processor. The misses are all due to compulsory misses when loading the cache line for the first time, and there are no evictions because there are no writes that cause invalidations.

4.1.5 ***No Sharing:*** All protocols exhibit the same pattern in this case because there is no sharing across processes that will cause MI

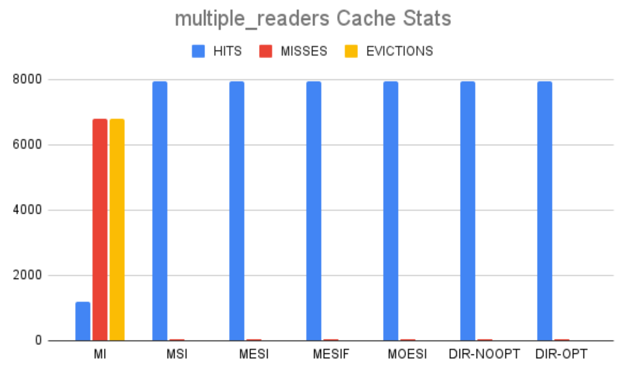

Figure 15: Multiple Reader: cache stats

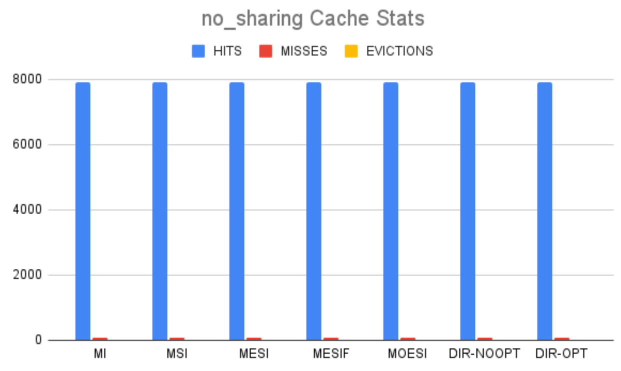

Figure 16: No sharing: cache stats

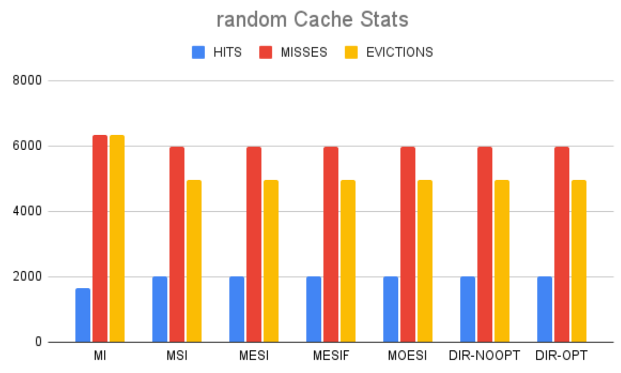

Figure 17: Random: cache stats

to differ from the rest. The rest exhibit similar patterns for reasons similar to previous benchmarks.

There are no evictions because the caches do not share data, and the hit rate is very high for similar reasons. The misses are all due to compulsory misses when loading the cache line for the first time.

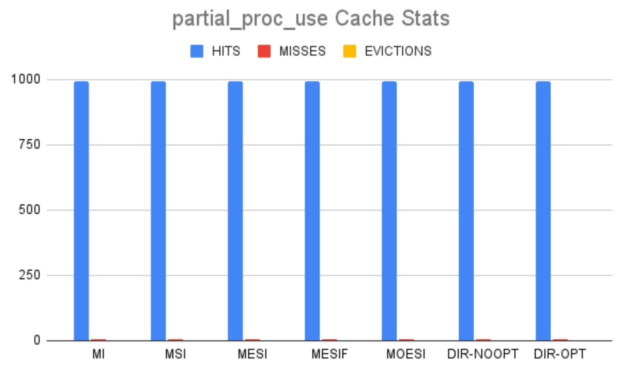

Figure 18: Partial Proc Use: cache stats

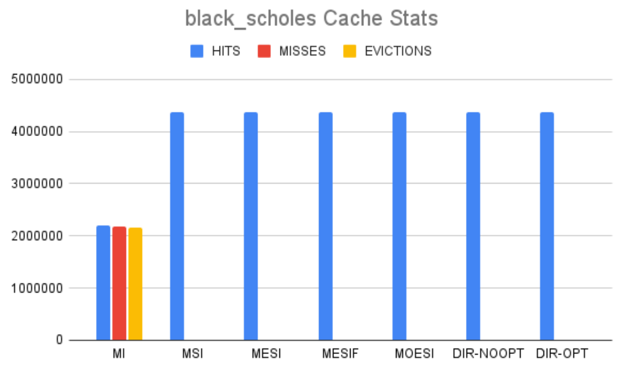

Figure 19: Black Scholes: cache stats

4.1.6 ***Partial Proc Use:*** These statistics are due to the single processor being used, resulting in no evictions, and the hits are very high. For this reason, the patterns across all coherence protocols are the same.

4.1.7 ***Black Scholes:*** There is a high amount of sharing between processors, so the difference between MI and the rest is significant. This results in MI having a significantly higher number of evictions.

There are a few writes to the shared cache line, and hence, a few evictions are recorded. The workload is embarrassingly parallel, so the reads mostly access a shared data structure to run some computations.

The evictions are the result of collecting the results after the computations, as the processors try to write the results to a data structure. These writes will cause invalidations.

4.2 Interconnect Utilization:

We further analyzed the interconnect traffic for the various workloads. For the synthetic traces, we generated workloads for 8 processors, while for the real life traces we used black scholes with 4 processors.

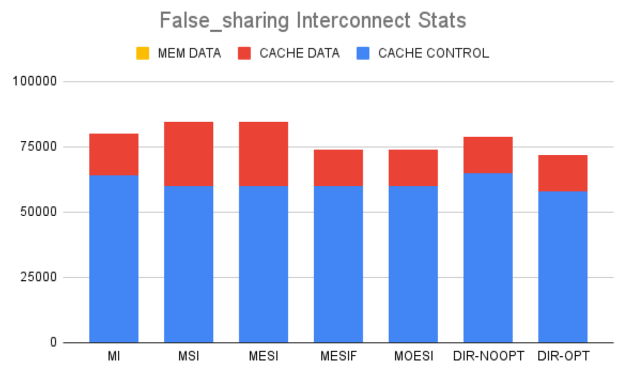

Figure 20: False Sharing: interconnect stats

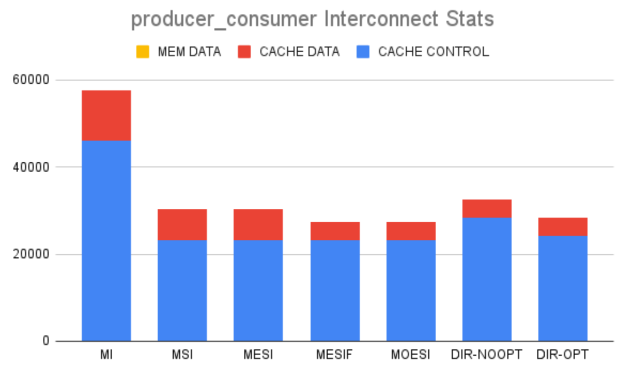

Figure 21: Producer Consumer: interconnect stats

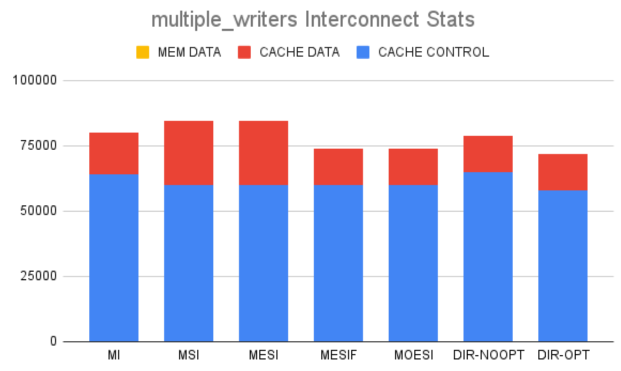

Figure 22: Multiple Writers: interconnect stats

4.2.1 ***False Sharing:*** Most of the interconnect traffic is due to cache control traffic because the majority of the messages are snooping request messages or directory request messages. The data provided by memory is not much because once the data has been provided by memory in the first miss, the caches provide the data for further requests from other caches.

MSI and MESI perform worse than MI. The traffic breakdown shows that they outperform MI in cache control traffic, but perform worse in cache data traffic. This is because multiple sharers provide the data whenever a new processor sends a read request for the cache line. MESIF and MOESI fix this problem, and they perform better than MI.

The cache control traffic portions between MESIF/MOESI and MSI/MESI are the same. However, the optimization is in the cache data traffic due to the Owner and forwarder optimizations in MOESI and MESIF, respectively. In these protocols, only one cache provides the data, as opposed to multiple caches providing the data when a read request is made for a line shared across multiple caches.

Dir-NOopt performs worse than MESIF/MOESI because the main advantage of directories is more pronounced when there are a large number of cores. At a small number of cores for some workloads, directory generates as much traffic or maybe even more than snooping.

Dir-Opt outperforms Dir-NOopt due to the intervention forwarding optimization. The cache data traffic between both are similar, but the optimization is in the cache control messages, as fewer directory messages are sent.

4.2.2 ***Producer Consumer:*** MI performs worse because of evictions caused without a Shared State. Every read will cause evictions in the processor that holds the line, and subsequent reads will cause a broadcast on the interconnect, rather than a hit.

MSI and MESI perform similarly because there is no Load-then-Store pattern within a processor's workload; therefore, the silent upgrades (E) in MESI do not occur here.

MESIF and MOESI perform similarly to and better than MSI/MESI because they have a designated cache that provides data when a read request is made for a cache line shared by multiple caches, as opposed to all the caches sharing the line providing the data.

Dir-NOopt performs worse than MESIF/MOESI, similar to what was observed in the previous section (low core count).

Dir-Opt outperforms Dir-NOopt, similar to what was observed in the previous section (intervention forwarding).

4.2.3 ***Multiple Writers:*** The traffic patterns here are similar to those for false sharing. The reasons for the patterns observed are also the same.

4.2.4 ***Multiple Readers:*** MI performs poorly here because the read advantage of other protocols over MI becomes more pronounced, especially when multiple processors share a line. The reasons for MI performing worse are the same as previously mentioned (no S state to share a line across multiple processors).

Zooming in on the traffic statistics without MI, MESI/MSI perform similarly for the reasons previously mentioned (no Load-then-Store within a processor's workload). MESIF/MOESI perform similarly and outperform MSI/MESI for the reasons previously mentioned (one cache provides data upon a read when multiple caches share the data).

However, Dir-NOopt outperforms MESIF/MOESI because most of the workload involves reading and there are no invalidations. Since there is a range in the number of readers (1-7), the directory avoids broadcasting snooping messages to processors that do

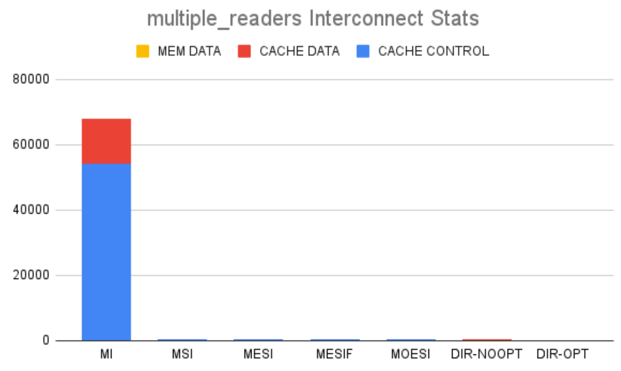

Figure 23: Multiple Readers: interconnect stats

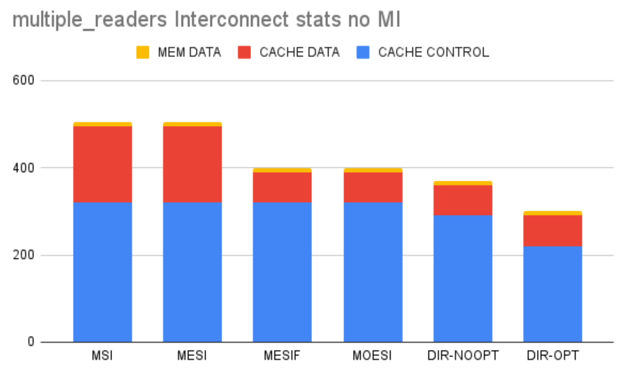

Figure 24: Multiple Readers No MI: interconnect stats

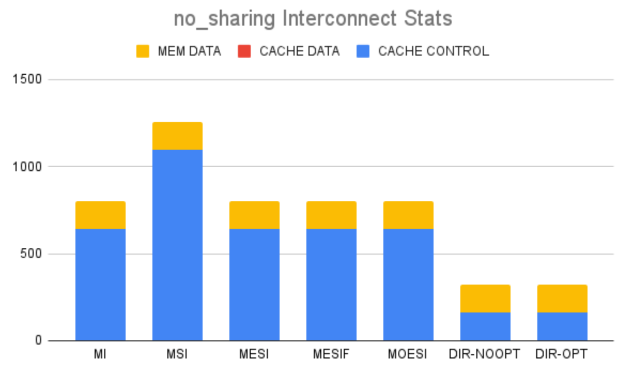

Figure 25: No Sharing: interconnect stats

not have the line locally. This is more pronounced for heavy read workloads.

Dir-Opt outperforms Dir-NOopt significantly because the benefit of intervention forwarding optimization affects reads.

4.2.5 ***No Sharing:*** Except for MSI, all snooping protocols perform similarly to each other because MSI generates additional traffic

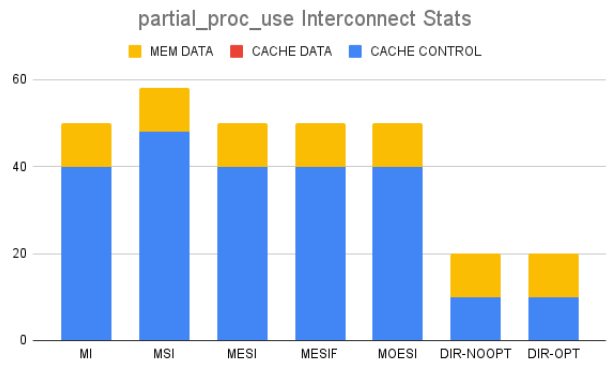

Figure 26: Partial Proc Use: interconnect stats

when transitioning from Shared State (S) to Modified State (M), even when there are no additional processes sharing the line. MESI fixes this with the silent upgrade, which allows the state to transition from Exclusive (E) to Modified (M) without generating additional bus traffic.

The directory protocols perform significantly better than snooping because they send unicast messages instead of broadcasting requests. In the absence of multiple caches sharing a line, the benefits of broadcasting are non-existent, so the directory performs much better in this case.

Both directory protocols perform the same because there are no shared lines between the processors, so there is no scenario that requires intervention forwarding optimization.

Since there is no sharing, caches do not provide any data, and all data is provided by memory.

4.2.6 *Partial Proc Use.* Here, one processor is used, and the other seven are idle.

MSI performs differently from the other snooping protocols for the same reasons as mentioned above.

Directory protocols perform significantly better than snooping because snooping still broadcasts the requests even though the other processors are not running the application. The benefits of directory and unicast messages as opposed to broadcasting really begin to shine here.

Since there is only one processor, caches do not provide any data, and all data is provided by memory.

4.2.7 ***Random:*** MSI/MESI are similar and perform better than MI due to the sharing optimization.

MESIF/MOESI performs better than MSI/MESI due to the data provided with sharing optimization.

Dir-NOopt performs worse than MESIF/MOESI for the reasons mentioned earlier (due to low core count behavior on non-sharing heavy workloads).

The workload is random and exhibits general expected trends.

4.2.8 ***Black Scholes:*** MI performs poorly here because the read advantage of other protocols over MI becomes more pronounced, since there is a lot of cache line sharing across multiple processors.

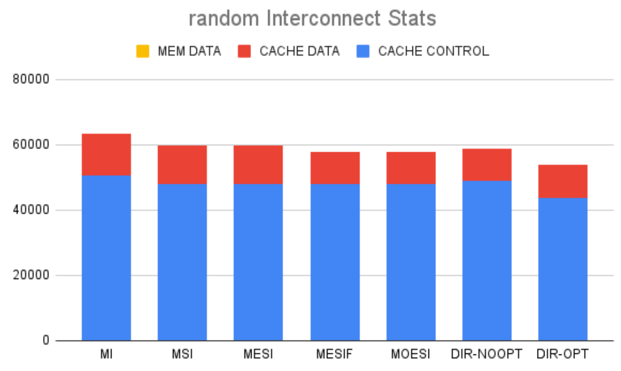

Figure 27: Partial Proc Use: interconnect stats

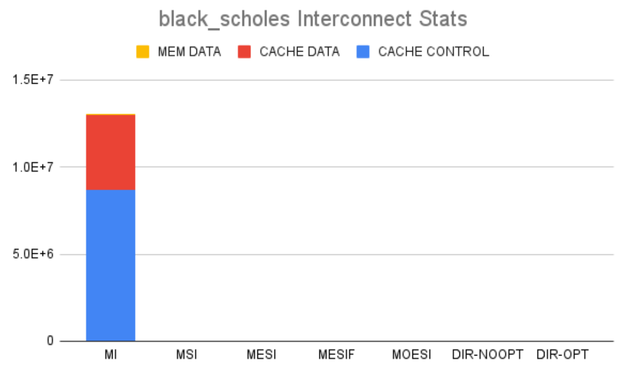

Figure 28: Black Scholes: interconnect stats

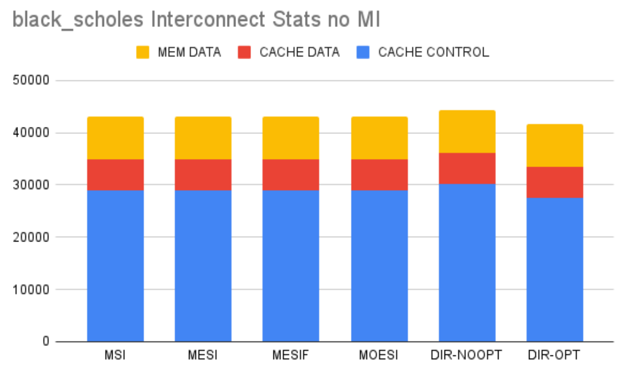

Figure 29: Black Scholes no MI: interconnect stats

The reasons for MI performing worse are the same as previously mentioned (no S state to share a line across multiple processors).

Zooming in on the traffic statistics without MI, the performance of all the protocols are very close, although MESIF/MOESI perform slightly better than MSI/MESI. This implies that for most of the computation, after sharing cache lines to read the inputs, the processors do not write to the inputs and instead perform their computation in parallel, requiring little to no cache line sharing for the majority of the computation.

4.3 Scalability with Core Count

During our evaluation, we also analyzed the traffic patterns when the core count was varied from 4 to 128 processors.

4.3.1 Snooping Protocols Scalability. In snooping-based protocols, as the number of processors increases, the traffic increases significantly. This is because snooping inherently relies on broadcasting messages to all processors. When analyzing the producer-consumer workload, we observed the following.

- MI protocol traffic increased from 804 to 831,610
- MSI protocol traffic increased from 606 to 369,578
- MESI protocol traffic increased from 606 to 369,578
- MESIF protocol traffic increased from 570 to 303,600
- MOESI protocol traffic increased from 574 to 303,871

The traffic scaling is approximately quadratic with processor count (34). This is because each request generates (n - 1) messages, where n is the number of processors. As n increases, the number of messages also grows quadratically.

We also consider the scalability with other traffic patterns:

False Sharing: In this case, as the processors write to different addresses within the same cache line, we end up with the worst case performance as shown in 30. Processors constantly invalidate each others' cache line despite accessing different data.

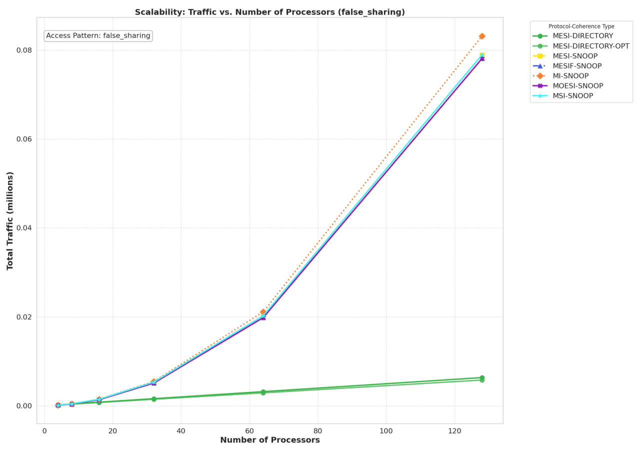

Figure 30: The scalability of snooping-based protocols using false sharing with processor count increase

Multiple-Writers: For the multiple writers workload, we observe similar patterns to false sharing. The MSI/MESI protocols actually scaled worse than MI (31) due to high number of write operations causing significant cache data traffic.

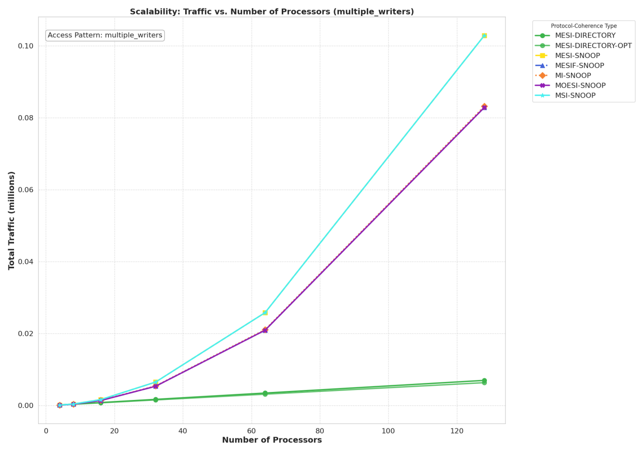

Figure 31: The scalability of snooping-based protocols using multiple writers with processor count increase

Multiple Readers: Read heavy workloads show the importance of more specialized coherence states like Forwarding and Owned states because those protocols have significantly reduced traffic due to the lesser amount of caches trying to send cache line data when requested by a particular cache (32).

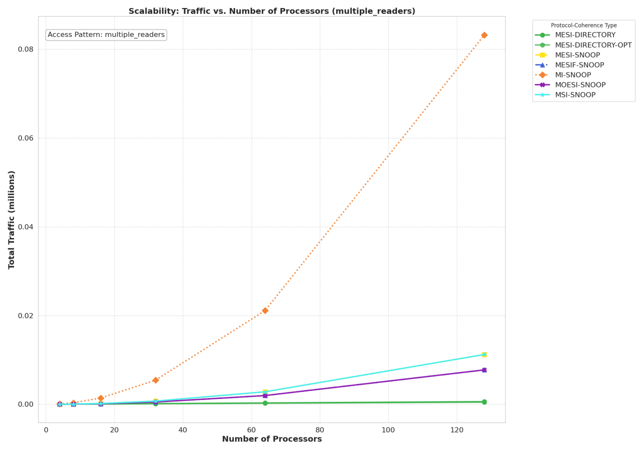

Figure 32: The scalability of snooping-based protocols using multiple readers with processor count increase

No Sharing: The no-sharing workload shows that snooping protocols' poor scalability isn't just due to coherence activity but the fundamental broadcasting mechanism itself. Even with no actual sharing, this scales quadratically as seen in 33.

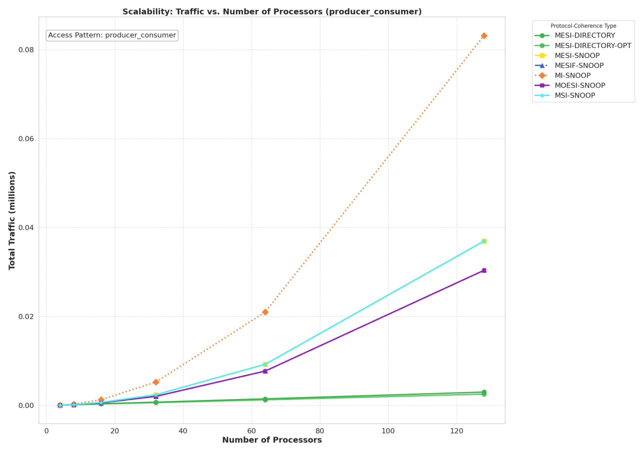

Figure 34: The scalability of snooping-based protocols with processor count increase

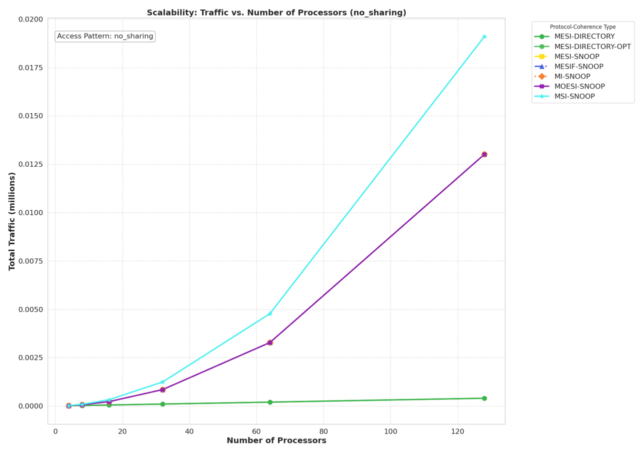

Figure 33: The scalability of snooping-based protocols using no sharing with processor count increase

Among snooping protocols, MOESI and MESIF consistently fare well and demonstrate better scaling than MI, MSI, and MESI due to their optimization for shared line access, but they still show the quadratic increase in terms of scalability.

4.3.2 Directory Protocol Scalability. Directory-based protocols show significantly better scaling. For the producer-consumer workload:

- Directory (without optimization) traffic increased from 1012 to 30,046.
- Directory (with optimization) traffic increased from 884 to 25,484.

The directory traffic scales more linearly (35) with the processor count because the directory sends targeted messages only to the processors that need them rather than broadcasting to all the processors.

In summary, the snooping based protocols show quadratic scaling while the directory based approaches are near-linear. We also note that the scalability is dependent on the underlying workload.

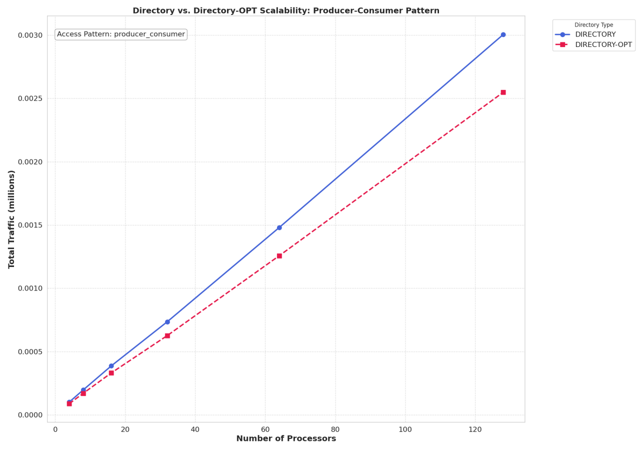

Figure 35: The scalability of directory-based protocols with processor count increase

5 Discussion

Based on our analysis in the previous section, we discuss some trade-offs/comparisons between the protocols.

5.1 Trade-Offs between Protocols

5.1.1 ***Snooping vs Directory.*** Snooping protocols do not scale well with traffic, unlike Directory, which scales well with traffic. However, at lower core counts, depending on the workload, the best snooping protocols can perform better than directory.

Snooping's advantage over directory is in the latency, which we did not capture in this project. As snooping is broadcast, all caches can get the messages at once, unlike directory where the directory has to get the message first before replying to the requestor with the intended cache[s]/forwarding the request on to the intended cache[s]

5.1.2 ***MSI vs MI.*** MSI's advantage over MI is allowing Sharers on a line, and this advantage is only noticed where there are multiple sharers on a line. For workloads that are write heavy without read sharing, MI and MSI's performance are very similar, and in some cases, MI actually outperforms MSI.

This happens when there are multiple sharers providing the data and there is no locality within the processor's workload to compensate for the additional traffic due to multiple sharer's providing the data.

5.1.3 ***MESI vs MSI.*** MESI's advantage over MI is when there are Load-Then-Store workloads with no evictions in between by other processors. In the absence of Load-Then-Store patterns without evictions between, the behavior of MESI and MSI are very similar.

5.1.4 ***MESI/MSI vs MI.*** MESI/MSI outperform MI when there are multiple sharers on a line and there is some locality within a processor's workload. In the absence of both of these conditions (e.g a write heavy workload), MI actually outperforms MSI/MESI.

5.1.5 ***MOESI vs MESIF.*** MOESI and MESIF performed the same in our analysis and we did not notice any significant differences between them. This makes sense since they solve the same problem of multiple sharers providing the data when a request is made for the line in different ways.

5.1.6 ***MOESI/MESIF over MESI.*** MOESI/MESIF's advantage over MESI is the designated cache that provides the data when a request is made to a line with multiple sharers. For workloads that do not have sharers, they perform similarly.

5.2 Conclusions

Based on our analysis we came up with the following conclusions

(1) Without multiple sharers to a cache line and some locality in processor's workloads, MSI and MESI perform worse than MI.
(2) At low core counts, Directory sometimes performs worse than best snooping protocols.
(3) At high core counts, Directory always outperforms even the best snooping protocols.
(4) Even with a single core workload on a multi-core system directory provides huge benefits.
(5) The main advantage of other protocols over MI is in reduced interconnect traffic and not in the hits/misses.
(6) All protocols except MI have the same hit/miss/eviction patterns, key difference is in the interconnect traffic reduced.
(7) The advantage of MOESI/MESIF over MESI/MSI is more pronounced when lots of caches are sharing a line.
(8) In the traffic optimizations, the cache control traffic shows the most variance across the different protocols, memory supplied data and cache supplied data are usually the same.

References

[1] 2025. Lecture 12: Snooping-based Cache Coherence. https://www.cs.cmu.edu/afs/cs/academic/class/15418-s25/public/lectures/12_cachecoherence1.pdf. https://www.cs.cmu.edu/afs/cs/academic/class/15418-s25/public/lectures/12_cachecoherence1.pdf 15-418/15-618: Parallel Computer Architecture and Programming, Carnegie Mellon University.
[2] 2025. Lecture 13: Directory-Based Cache Coherence. https://www.cs.cmu.edu/afs/cs/academic/class/15418-s25/public/lectures/13_directorycoherence.pdf. https://www.cs.cmu.edu/afs/cs/academic/class/15418-s25/public/lectures/13_directorycoherence.pdf 15-418/15-618: Parallel Computer Architecture and Programming, Carnegie Mellon University.
[3] Somdip Dey and Mamatha S. Nair. 2014. Design and Implementation of a Simple Cache Simulator in Java to Investigate MESI and MOESI Coherency Protocols. *International Journal of Computer Applications* 87, 11 (February 2014). doi:10.5120/15188-3531
[4] Marek Fiser. 2014. MESIF Protocol. https://texample.net/mesif/ Accessed: 2025-04-28.
[5] Brian P. Railing. 2024. CADSS: Computer Architecture Design Simulator for Students. In *Proceedings of the Workshop on Computer Architecture Education* (Orlando, FL, USA) *(WCAE '23)*. Association for Computing Machinery, New York, NY, USA, 34–40. doi:10.1145/3605507.3610626