Parallel Architectures: Assignment 2

Adam McCarthy (s1579267)

February 19, 2016

1 Introduction

This assignment involves the development of a cache coherence simulator for the MSI and MESI invalidation-based protocol and the MES update-based protocol. Section 2 describes the implementation of the simulator. Section 3 then describes the system testing used to validate that the implementation of the simulator matches the specification. Finally, Section 4 reviews the experiments conducted and their results.

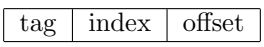
2 Implementation of the simulator

The simulator has been implemented in Python $\geq$ 2.7. As there are few external dependencies, using the *pypy* interpreter results in an approximately 10x speedup. The primary components are as follows:

- `Cache` A base class which implements a finite state machine, given a set of state transitions.
- `MSICache`, `MESICache`, `MESCache` Classes which inherit `Cache` and specify the state transitions for that protocol, along with any bespoke functionality.
- `Bus` An atomic bus which accepts bus transactions and allows caches to query and update the state of other caches.

2.1 Mapping addresses to the cache

The cache is composed of a number of cache sets (rows) where each row has a certain block size. In our case, word addressing is used, so if the block size is four then each cache set will contain four words. Given an address in binary, we split the digits as follows:



If the block size is B then $log_2 B$ bits will be needed to specify the offset. Similarly, if the number of cache sets is S then $log_2 S$ bits will be needed to specify the index of the cache set. As the cache is smaller than the main memory, it is possible for collisions to occur. The remaining bits, termed tag, are stored with the cache set at index and can be compared later to detect collisions.

When data is stored in the cache, the offset bits are set to 0 to represent the start of the data in memory and B words from that point onwards are then copied into the cache. The offset can then be used later to extract the data at the referenced address.

2.2 `Cache`

This class takes a set of state transitions as a nested dictionary (hash table) in the form

$$[op][is\_me][current\_state] \rightarrow new\_state \tag{1}$$

where *op* is the operation performed (R or W), *is_me* is a Boolean flag representing whether the bus transaction is related to the current cache and *current_state* is the current state of the line upon which the transaction has been performed.

Additionally, this class records statistics and updates the tag stored in a cache line in the event of a miss. Finally, it sets the initial state of all blocks to be a defined *reset_state*.

2.3 `MSICache`

The *reset_state* for the MSI protocol is I and state transitions are defined as follows in Python:

```
{"R": {True: {"I": "S",
              "S": "S",
              "M": "M"},
       False: {"M": "S",
               "S": "S",
               "I": "I"}},
 "W": {True: {"I": "M",
              "S": "M",
              "M": "M"},
       False: {"S": "I",
               "M": "I",
               "I": "I"}}}
```

2.4 `MESICache`

The *reset_state* for the MESI protocol is I and the state transactions are defined as follows in Python:

```
{"R": {True: {"I": "SE",
              "S": "S",
              "M": "M",
              "E": "E"},
       False: {"M": "S",
               "E": "S",
               "S": "S",
               "I": "I"}},
 "W": {True: {"I": "M",
              "S": "M",
              "M": "M",
              "E": "M"},
       False: {"S": "I",
               "M": "I",
               "I": "I",
               "E": "I"}}}
```

The SE state here is not a full transient state in that it does not remain after the end of the current cycle. It simply identifies that there is additional work to be done by the cache before the end of the cycle.

If the cache sees that the state for the current index is SE, it queries to bus to determine whether any other caches have the current index in states E, S or M. If they do it sets the state for itself and those caches to be S. Otherwise, it sets its own state to be E.

2.5 MESCache

The *reset_state* for the MES protocol is *None* and the state transitions are defined as follows in Python:

```
{"R": {True: {"E": "E",
              "S": "S",
              "M": "M",
              None: None},
       False: {"M": "S",
               "E": "S",
               "S": "S",
               None: None}},
 "W": {True: {"E": "M",
              "M": "M",
              "S": "S",
              None: None},
       False: {"S": "S",
               "E": "S",
               "M": "S",
               None: None}}}
```

If the bus transaction is related to this cache, further processing is performed. If the transaction resulted in the hit and is a write in the S state, an write-update message must be sent. As we have an atomic bus, the cache line in all other caches are updated to contain the current tag and their states are updated to be S.

Otherwise, if the transaction resulted in a miss and there is another cache in state E, S or M, update the state to be S. However, if no other cache has this line in state E, S or M, set the state to be E for a read and M for a write.

2.6 `Bus`

`Bus` is a simple class with an array containing the caches in the system. A reference to the bus is also passed into each cache when it is initialised. This allows all caches to query the state of all other caches and update them if needed. The `Bus` also receives lines from the trace file and either broadcasts them to all the caches or prints statistics as appropriate.

A more complex multiprocess implementation using transient states and a message bus where caches could not directly query each other was also completed, however, upon advice a simpler, single process implementation was submitted.

2.7 Reporting

The reporting operations, v, p, h and i have been implemented as per the original specification. p also takes an optional address argument to only print the state for that address. The following metrics are collected and stored in a serialised format for later analysis:

- Read hits for all data
- Read hits for private data
- Read misses
- Total read transactions
- Write hits for all data
- Write hits for private data
- Write misses
- Total write transactions
- Invalidation broadcasts
- Number of cache lines invalidated due to invalidation broadcasts
- Write-update messages

- Number of cache lines updated due to write-update messages
- Write-back messages

3 Validation of the simulator

To validate the simulator, a small test framework was written which executes trace files and compares the final state to an expected state. To define the expected state, a comment operator was added to the trace files. The following comment defines an end state, where SX defines the state for that cache. There can be multiple end states defined.

```
# EndState: <address> <S1>,<S2>,<S3>,<S4>
```

The test suite can then be run as follows:

```
$ nosetests
..............
Ran 14 tests in 0.028s

OK
```

The following tests were performed:

Protocol	State	Transaction
MSI	S	Read Miss

`# EndState: 2 S,S,I,I`
`P0 R 2`
`P1 R 2`		

MSI	S	Write Miss

`# EndState: 2 M,I,I,I`
`P0 R 2`
`P1 R 2`
`P0 W 2`		

MSI	M	Read Miss

`# EndState: 2 S,S,I,I`
`P0 W 2`
`P1 R 2`		

MSI	M	Write Miss

```
# EndState: 2 I,M,I,I
P0 R 2
P1 R 2
P0 W 2
P1 W 2
```

MESI	M	Read Miss

```
# EndState: 2 S,S,I,I
P0 R 2
P0 W 2
P1 R 2
```

MESI	Not S	Read Miss

```
# EndState: 2 E,I,I,I
P0 R 2
```

MESI	S	Read Miss

```
# EndState: 2 S,S,I,I
P0 R 2
P1 R 2
```

MESI	S	Write

```
# EndState: 2 M,I,I,I
P0 R 2
P1 R 2
P0 W 2
```

MESI	E	Write

```
# EndState: 2 M,I,I,I
P0 R 2
P0 W 2
```

MES	Not S	Read Miss

```
# EndState: 2 E,None,None,None
P0 R 2
```

MES	Not S	Write Miss

```
# EndState: 2 M,None,None,None
P0 W 2
```

MES	S	Read Miss

```
# EndState: 2 S,S,None,None
P0 R 2
P1 R 2
```

MES	S	Write Miss

```
# EndState: 2 S,S,None,None
P0 W 2
P1 W 2
```

MES	E	Write Hit

```
# EndState: 2 M,None,None,None
P0 R 2
P0 W 2
```

4 Design and execution of experiments

These experiments can be repeated as follows:

```
pypy main.py --msi --mesi --mes <trace filename> --record
python plot_graphs.py
```

4.1 Compare the behaviour of the three protocols in terms of miss-rate and messages sent on the bus

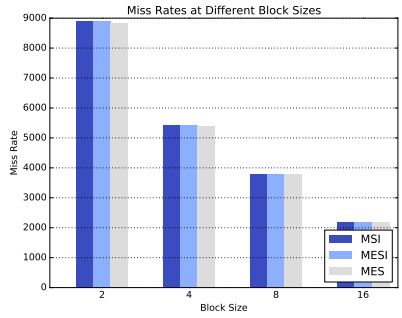


Figure 1: Trace 1

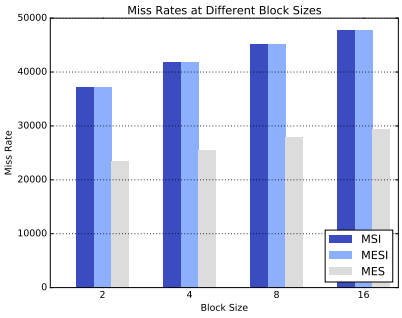
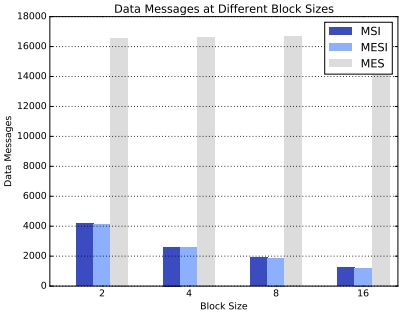
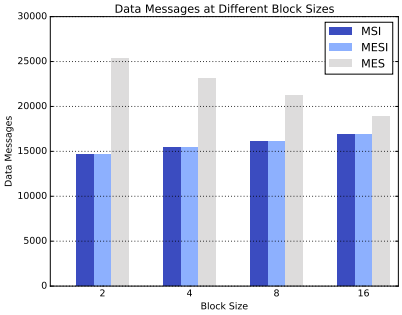


Figure 2: Trace 2





Across both example workloads, the update-based protocol had a lower miss rate. With invalidation-based protocols, other caches miss after a write because the block has been invalidated, whereas with update-based protocols they don't. However, it had a significantly higher number of messages sent on the bus. This is because there may be multiple-useless copies if all processors aren't working on the same data.

Comparing the invalidation-based protocols, MSI and MESI, they had the same miss rates. This is to be expected as including the E state does not affect the data stored in the caches. However, as expected, MESI had a slightly lower number of messages sent on the bus than MSI. This is because in MSI it takes two bus transactions to read and modify data ($I \rightarrow S$ and $S \rightarrow M$) whereas the MESI protocol only requires one ($I \rightarrow E$) if no other cache has a copy.

4.2 How do differing cache-line/block sizes affect miss/hit rates and invalidations/updates?

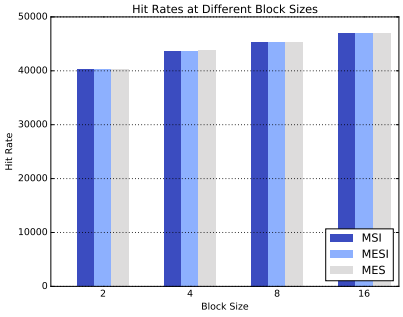


Figure 3: Trace 1

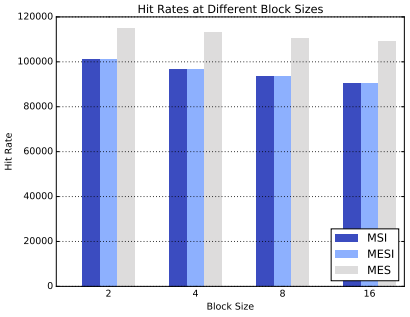
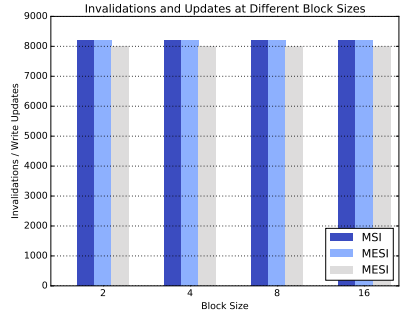
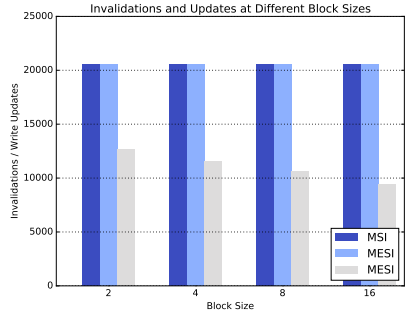


Figure 4: Trace 2





The hit rate for trace 1 is increasing as the block size is increasing. This implies that there is a low level of coherence misses using this workload (spatial locality is good) and therefore the rate of true sharing misses is decreasing. The hit rate for trace 2 is decreasing, however. This implies that there is a high degree of false sharing in trace 2 due to poor spatial locality.

4.3 In terms of memory accesses, what is the distribution of accesses to private data and shared data?

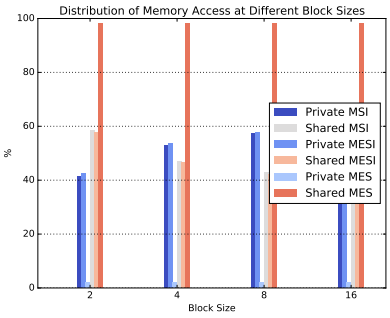


Figure 5: Trace 1

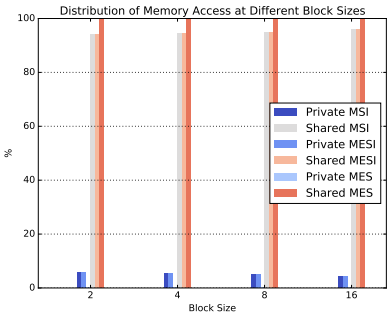


Figure 6: Trace 2

As expected, update-based protocols spend more time in a Shared state than invalidation-based protocols. This is because on a write, update-based protocols send update messages which cause other caches to move into a Shared state. Invalidation-based protocols, however, cause other caches to move to an Invalid state where they stay until they next access the block.

Trace 1 spends a much higher proportion of the time accessing private data. This matches the result in the previous section where we discovered a low level of coherence misses. Trace 2, however, spends significantly more time accessing Shared data, which matches out result in the previous section where we discovered a high degree of false sharing.

5 Conclusion

This report has shown that although update-based protocols can reduce the rate of cache misses, they send a significantly higher number of messages on the bus. The bus generally has a limited bandwidth which should ideally be used for memory access, so it is often advantageous to limit its use. Update-based protocols are also more complex than invalidation-based protocols in their use of the bus.

The introduction of the Exclusive state in the MESI protocol slightly reduces the number of messages sent on the bus, while maintaining the same properties as the MSI protocol.

We have also shown that maintaining spatial locality is key to the high performance of these protocols. Increasing the block size is only beneficial if this is maintained. This problem cannot be addressed in hardware design, however, rather it should be addressed during compilation. There is likely a critical point where further increasing the block size is not beneficial due to the risk of false sharing misses.

Finally, we have shown that the greater the proportion of private data access, the higher the performance. This reinforces the importance of limiting shared data access across threads and processes, and also the importance of controlling memory access during compilation.