## Description of internal structure and workings of simulator

- The “CacheLine” class represents a single line in a cache, and maintains a state for that line, as well as a dictionary containing all of the tags (represented by binary strings) in that line.
- The “Cache” class represents the cache of one CPU, and this maintains counts of various properties during a simulation: the amount of hits in this cache, the amount of data messages generated by this cache, and so on. It generates an an array of “CacheLine” objects when instantiated, defaulting them to empty and invalid state if the cache is in MSI/MESI mode, or empty and in no state if the cache is in MES mode.
  - The “load_line” function simulates loading a full line into the cache.
  - The “access” function simulates an operation occurring and accessing the cache. Various helper functions are invoked (for instance “msi_m_rw_hit” is invoked on a cache in MSI mode, with the relevant line in modified state, experiencing a read/write hit) to accomplish this, and a message is returned containing information about the behaviour of the cache during the access.
  - The “print_X” functions handle printing various statistics to the console.
  - The “split_address” function breaks an incoming address into index, offset, and tag.
  - The “tag_check” function is used to check if a tag exists in a cache line given an index and offset (probably deprecated).
  - The “only_copy” function is used to check if the current cache is maintaining the only copy of a piece of memory, analogous to the “SHARED/NOT SHARED” condition.
- The “Simulator” class sets up four instances of “Cache” dependent on given parameters.
  - The “run” function iterates over the lines given in a file, executing the relevant code for each line. If the line-by-line/verbose option has been enabled, it parses the data returned by each cache’s “access” function, and outputs a message. In addition, at the end of a run, it outputs a summary of the run (overall hit rates, and so on).
- The “check_args” and “print_usage” functions are for parsing command-line arguments and informing the user how to correctly input them, respectively.
- The “get_bin_exponent” function is used to find binary exponents, useful for splitting addresses into index, offset, and tag.
- A list of assumptions made in the implementation of the simulator is available in the Appendix

The simulator, to the best of my knowledge, is fully complete.

## Validation of simulator

Several test files were created, to attempt to validate the simulator. These were simply run with built-in statistic outputting switches enabled, in order to manually compare the simulator against expected results. A simple test script (“test.sh”) was created, and could be run with no arguments to test all three cache protocols, or with “MSI/MESI/MES” as an argument to test a specific protocol. In addition, using “#” to comment lines in a test file was added to aid testing.

All tests functioned as expected, and produced the events described. The exact ordering of operations described at each step is irrelevant, as the simulation is only concerned with the workings of the cache, not how memory would be updated in practice, and so on.

*test_0.trace*

```
01.  # MSI: write to cache block in state shared in another cache
02.  # MESI: write to cache block in state exclusive in another cache
03.  # MES: write to cache block in state exclusive in another cache
04.  v
05.  P0 R 0
06.  p
07.  i
08.  o
09.  P1 W 0
10.  p
11.  i
12.  o
13.  h
```

MSI: write to cache block in state shared in another cache
5 - CPU0 now contains address 0 in shared state.
9 - CPU1  now contains address 0 in modified state. CPU1 sends 1 invalidation broadcast. CPU0 invalidates address 0 (1 invalidation).
Final result: CPU1 contains address 0 in modified state; 1 invalidation at CPU0; 1 invalidation broadcast from CPU1.

MESI: write to cache block in state exclusive in another cache
5 - CPU0 now contains address 0 in exclusive state.
9 - CPU1  now contains address 0 in modified state. CPU1 sends 1 invalidation broadcast. CPU0 invalidates address 0 (1 invalidation).
Final result: CPU1 contains address 0 in modified state; 1 invalidation at CPU0; 1 invalidation broadcast from CPU1.

MES: write to cache block in state exclusive in another cache
5 - CPU0 now contains address 0 in exclusive state.

9 - CPU1  now contains address 0 in shared state. CPU1 sends 1 write-update broadcast. CPU0 updates address 0 (1 update).
Final result: CPU0 and CPU1 contain address 0 in shared state; 1 update at CPU0; 1 update broadcast from CPU1.

*test_1.trace*

```
01.   # MSI: read to cache block in state modified in another cache
02.   # MESI: read to cache block in state modified in another cache
03.   # MES: read to cache block in state modified in another cache
04.   v
05.   P0 W 0
06.   p
07.   i
08.   o
09.   P1 R 0
10.   p
11.   i
12.   o
13.   h
```

MSI: read to cache block in state modified in another cache
5 - CPU0 now contains address 0 in modified state. CPU0 sends 1 invalidation broadcast.
9 - CPU0 writes back address 0, and transitions to shared state. CPU1 now contains address 0 in shared state.
Final result: CPU0 and CPU1 contain address 0 in shared state; 1 invalidation broadcast from CPU0; 1 write-back from CPU0.

MESI: read to cache block in state modified in another cache
5 - CPU0 now contains address 0 in exclusive state. CPU0 sends 1 invalidation broadcast.
9 - CPU0 writes back address 0, and transitions to shared state. CPU1 now contains address 0 in shared state.
Final result: CPU0 and CPU1 contain address 0 in shared state; 1 invalidation broadcast from CPU0; 1 write-back from CPU0.

MES: read to cache block in state modified in another cache
5 - CPU0 now contains address 0 in exclusive state. CPU0 sends 1 invalidation broadcast.
9 - CPU0 writes back address 0, and transitions to shared state. CPU1 now contains address 0 in shared state.
Final result: CPU0 and CPU1 contain address 0 in shared state; 1 write-back from CPU0.

*test_2.trace*

```
01.   # MSI: read to cache block in state shared in another cache
```

```
02.  # MESI: read to cache block in state exclusive in another
     cache
03.  # MES: read to cache block in state exclusive in another
     cache
04.  v
05.  P0 R 0
06.  p
07.  i
08.  o
09.  P1 R 0
10.  p
11.  i
12.  o
13.  h
```

MSI: read to cache block in state shared in another cache
5 - CPU0 now contains address 0 in shared state.
9 - CPU1 now contains address 0 in shared state.
Final result: CPU0 and CPU1 contain address 0 in shared state.

MESI: read to cache block in state exclusive in another cache
5 - CPU0 now contains address 0 in exclusive state.
9 - CPU1 now contains address 0 in shared state. CPU0 switches address 0 to shared state.
Final result: CPU0 and CPU1 contain address 0 in shared state.

MES: read to cache block in state exclusive in another cache
5 - CPU0 now contains address 0 in exclusive state.
9 - CPU1 now contains address 0 in shared state. CPU0 switches address 0 to shared state.
Final result: CPU0 and CPU1 contain address 0 in shared state.

*test_3.trace*

```
01.  # MSI: write to cache block in state modified in another
     cache
02.  # MESI: write to cache block in state modified in another
     cache
03.  # MES: write to cache block in state modified in another
     cache
04.  v
05.  P0 W 0
06.  p
07.  i
08.  o
09.  P1 W 0
```

```
10.  p
11.  i
12.  o
13.  h
```

MSI: write to cache block in state modified in another cache
5 - CPU0 now contains address 0 in modified state. CPU0 sends 1 invalidation broadcast.
9 - CPU1 now contains address 0 in modified state. CPU1 sends 1 invalidation broadcast. CPU0 invalidates address 0 (1 invalidation).
Final result: CPU1 contains address 0 in modified state; 1 invalidation broadcast from CPU0; 1 invalidation broadcast from CPU1; 1 invalidation at CPU0.

MESI: write to cache block in state modified in another cache
5 - CPU0 now contains address 0 in modified state. CPU0 sends 1 invalidation broadcast.
9 - CPU1 now contains address 0 in modified state. CPU1 sends 1 invalidation broadcast. CPU0 invalidates address 0 (1 invalidation).
Final result: CPU1 contains address 0 in modified state; 1 invalidation broadcast from CPU0; 1 invalidation broadcast from CPU1; 1 invalidation at CPU0.

MES: write to cache block in state modified in another cache
5 - CPU0 now contains address 0 in modified state.
9 - CPU1 now contains address 0 in shared state. CPU1 sends a write-update message. CPU0 writes back address 0, updates its copy of address 0, and transitions to shared state.
Final result: CPU0 and CPU1 contain address 0 in shared state; 1 update at CPU0; 1 write-update message from CPU1; 1 write-back from CPU0.

*test_4.trace*

```
01.  # MESI: read to cache block in state shared in another cache
02.  # MES: read to cache block in state shared in another cache
03.  v
04.  P0 R 0
05.  P1 R 0
06.  p
07.  i
08.  o
09.  P2 R 0
10.  p
11.  i
12.  o
13.  h
```

MESI: read to cache block in state shared in another cache
4 - CPU0 now contains address 0 in exclusive state.

5 - CPU1 now contains address 0 in shared state. CPU0 transitions address 0 to shared state.
9 - CPU2 now contains address 0 in shared state.
Final result: CPU0, CPU1, and CPU2 contain address 0 in shared state.

MES: read to cache block in state shared in another cache
4 - CPU0 now contains address 0 in exclusive state.
5 - CPU1 now contains address 0 in shared state. CPU0 transitions address 0 to shared state.
9 - CPU2 now contains address 0 in shared state.
Final result: CPU0, CPU1, and CPU2 contain address 0 in shared state.

*test_5.trace*

```
01.   # MESI: write to cache block in state shared in another cache
02.   # MES: write to cache block in state shared in another cache
03.   v
04.   P0 R 0
05.   P1 R 0
06.   p
07.   i
08.   o
09.   P2 W 0
10.   p
11.   i
12.   o
13.   h
```

MESI: write to cache block in state shared in another cache
4 - CPU0 now contains address 0 in exclusive state.
5 - CPU1 now contains address 0 in shared state. CPU0 transitions address 0 to shared state.
9 - CPU2 now contains address 0 in modified state. CPU2 sends an invalidation broadcast. CPU0 and CPU1 invalidate their copies of address 0.
Final result: CPU2 contains address 0 in modified state; 1 invalidation broadcast from CPU2; 1 invalidation at CPU0; 1 invalidation at CPU1.

MES: write to cache block in state shared in another cache
4 - CPU0 now contains address 0 in exclusive state.
5 - CPU1 now contains address 0 in shared state. CPU0 transitions address 0 to shared state.
9 - CPU2 now contains address 0 in shared state. CPU2 sends a write-update message. CPU0 and CPU1 update their copies of address 0.
Final result: CPU0, CPU1, and CPU2 contain address 0 in shared state; 1 write-update broadcast from CPU2; 1 update at CPU0; 1 update at CPU1.

## Experiments and discussion of results

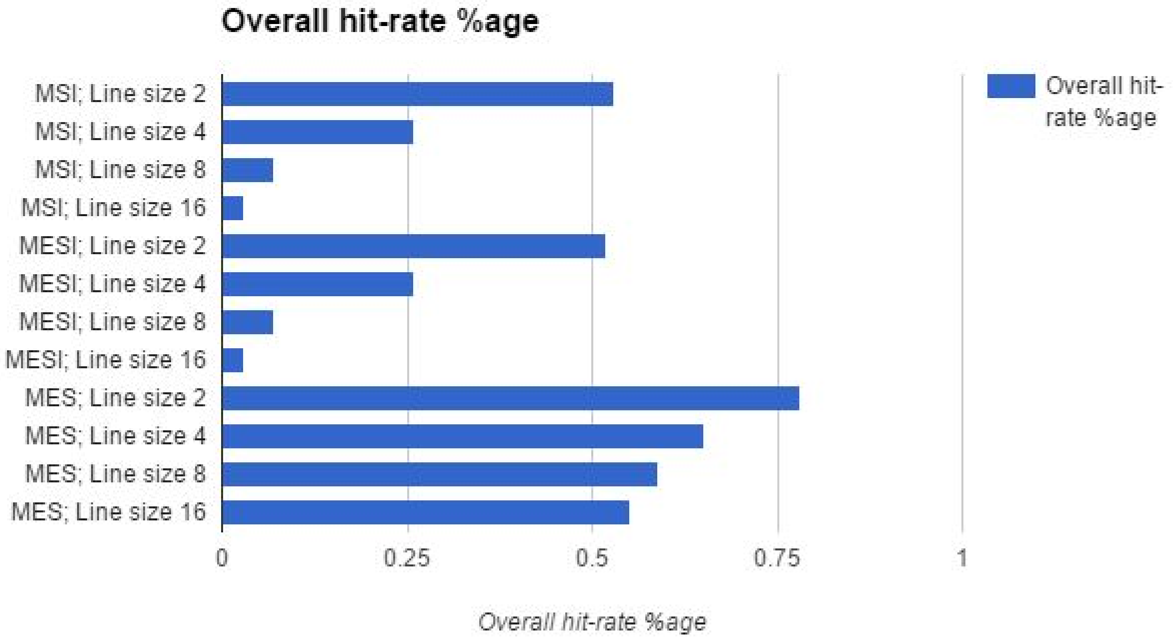
The experiments were run using a similar script to that for testing ("run.sh"). As before, this could be executed with no arguments, to run the complete simulation set, or executed with one argument of MSI/MESI/MES to run the simulations for a specific protocol.

**Hit rate:**

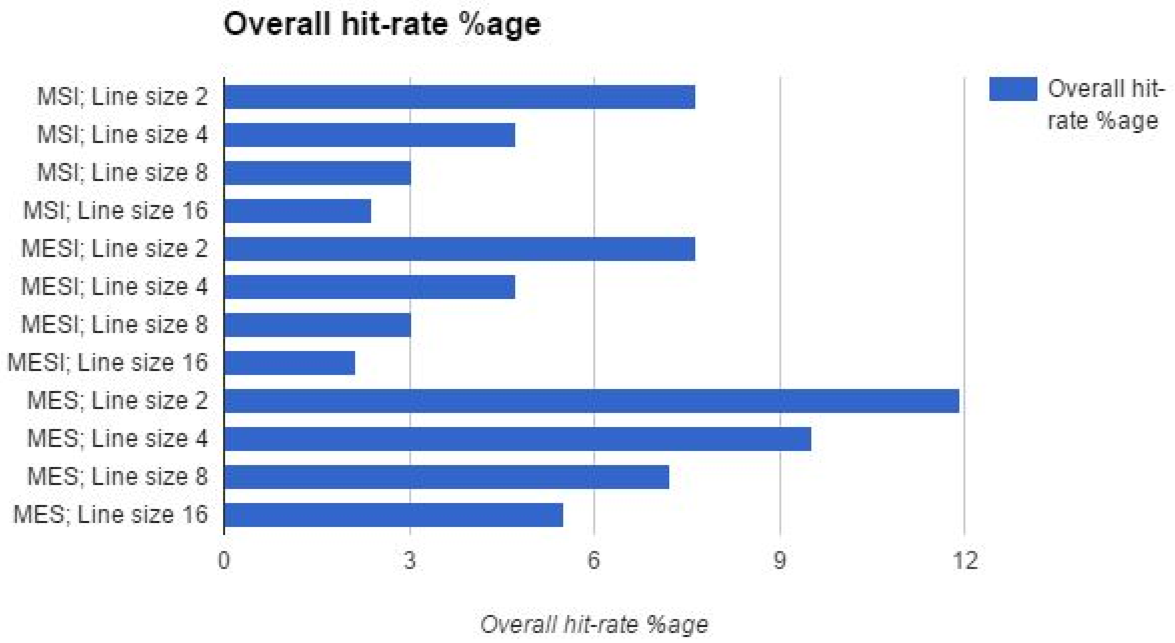
The hit-rate for the MSI and MESI protocols in trace 1 was very similar, with the MES protocol exhibiting a (relatively) much higher hit-rate. This is likely due to the invalidation taking place on a write to a shared line in the MSI and MESI protocols, where every other copy of a line will be invalidated upon one copy updating, compared to the MES protocol which instead updates any other copies held by other caches. All protocols suffered a reduction in hit-rate as the line size increased, likely due to a combination of capacity, conflict, and coherence misses.

For trace 2, a similar set of results was observed.

*Trace 1*



*Trace 2*

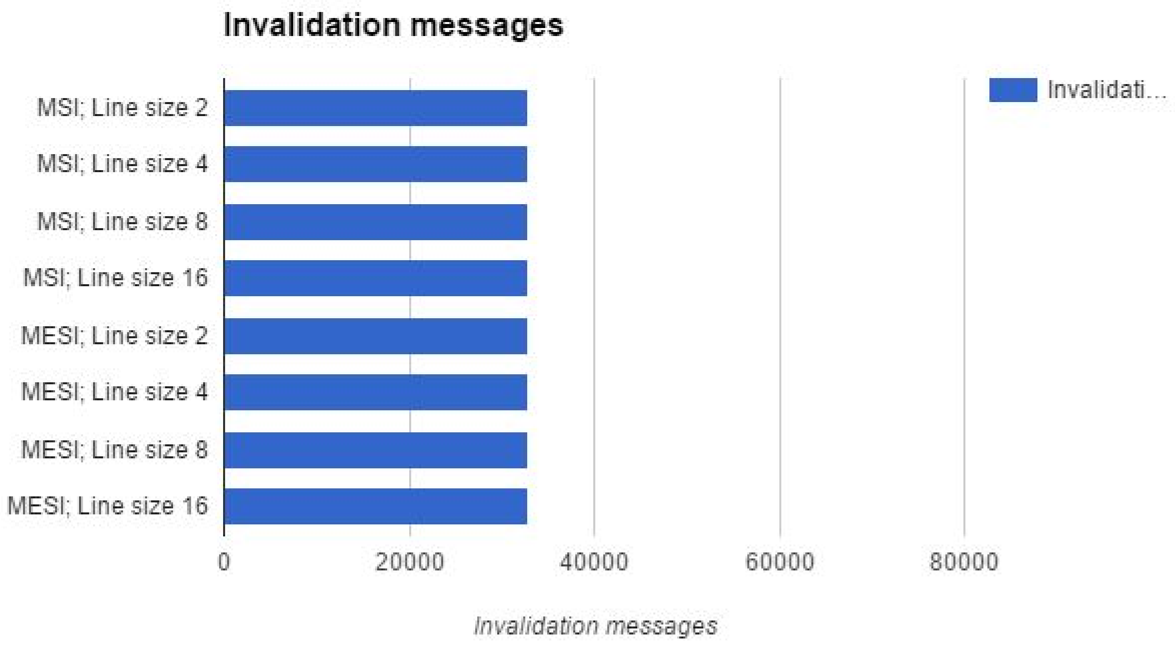


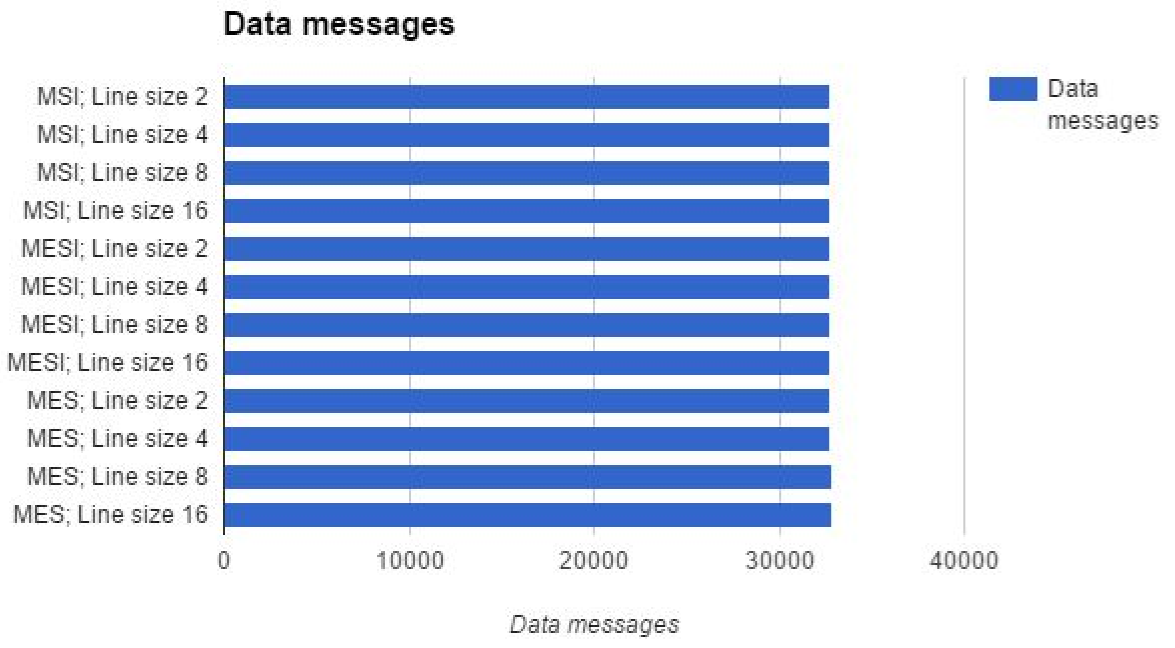
**Messages sent on the bus:**

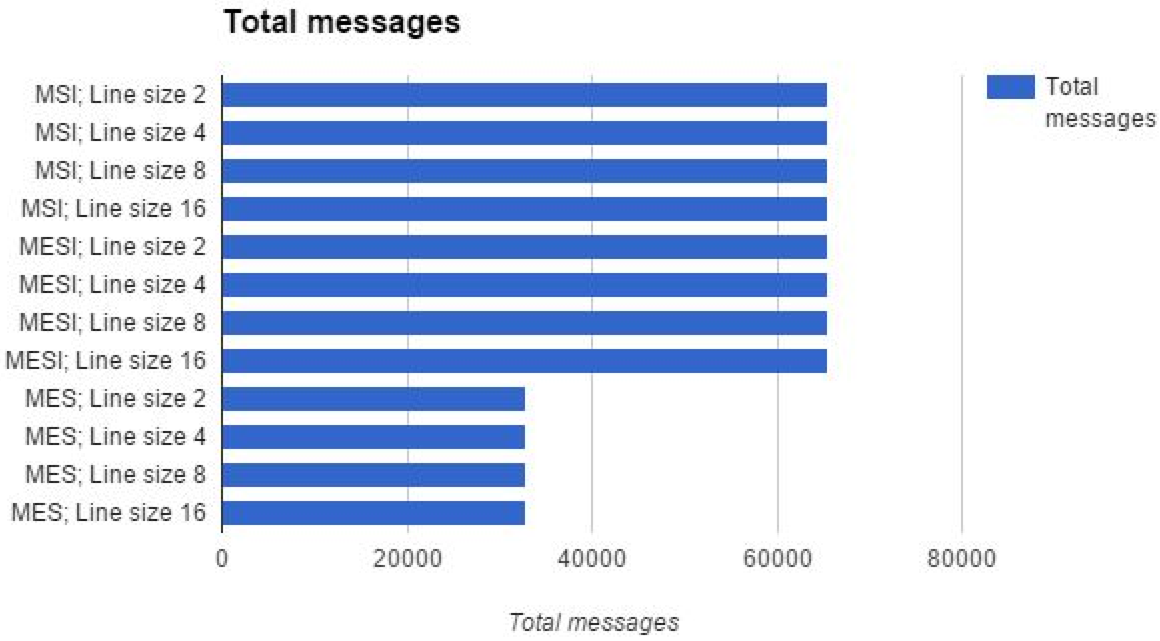
In trace 1, the number of invalidation messages sent on the bus was very similar between the MSI and MESI protocols. Additionally, the number of data messages sent on the bus for all protocols was very similar. The MES protocol therefore had far fewer total messages sent due to the protocol not utilizing invalidation messages.

In trace 2, the number of invalidation messages increased with the line size of the cache (inversely relational to the cache size). The MESI protocol had fewer invalidation messages, due to the use of the exclusive state, where an invalidation broadcast is not required to transition from exclusive to modified. The MES protocol sends more data messages than either the MSI or the MESI protocol, due to a write-update based nature of the protocol. Due to the lower hit-rate in higher line size MES caches, there are fewer write-update messages sent. This is due to a higher eviction rate, resulting in shared lines (which, when written to, would incur a write-update message) being removed from the cache, in favour of loading other lines which could be in modified or exclusive state, requiring no write-update message to be written to. The overall message count still favours the MES protocol, again due to a lack of invalidation messages.

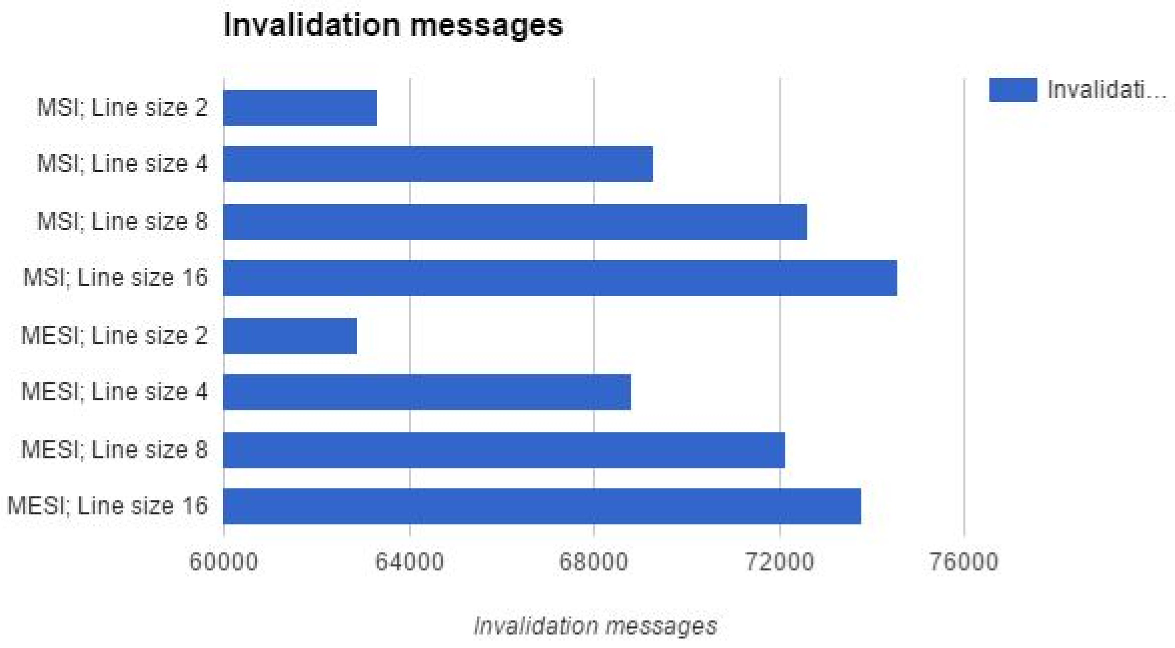
*Trace 1:*

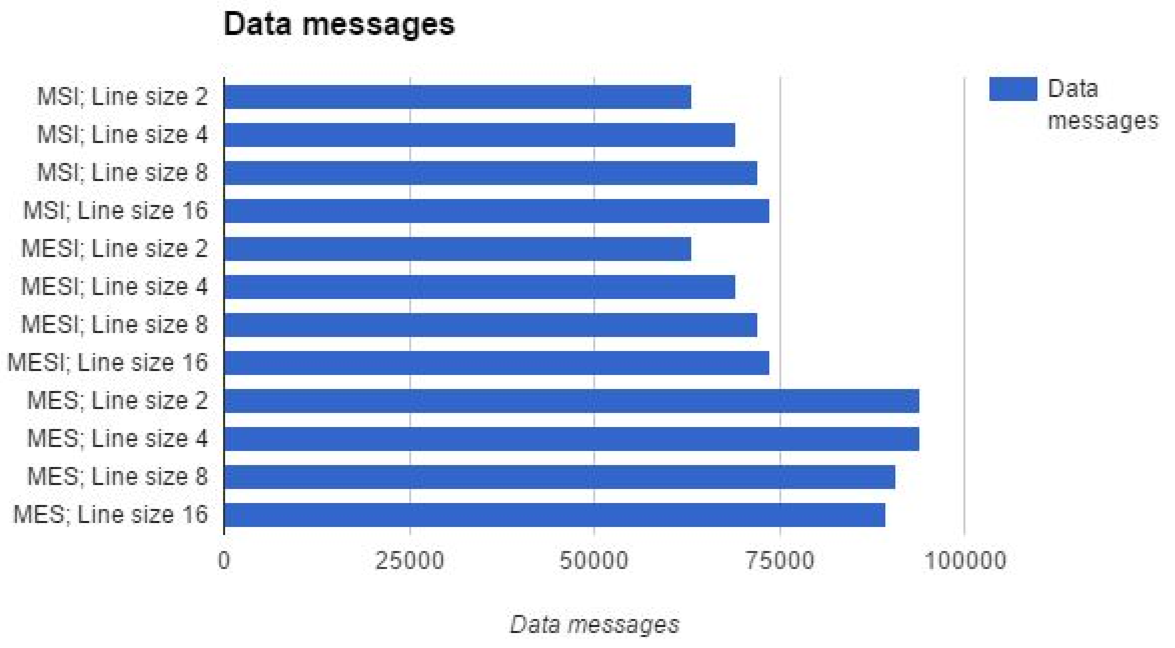

Invalidation messages
MSI; Line size 2
MSI; Line size 4
MSI; Line size 8
MSI; Line size 16
MESI; Line size 2
MESI; Line size 4
MESI; Line size 8
MESI; Line size 16
0
20000
40000
60000
80000
Invalidation messages
Invalidati...

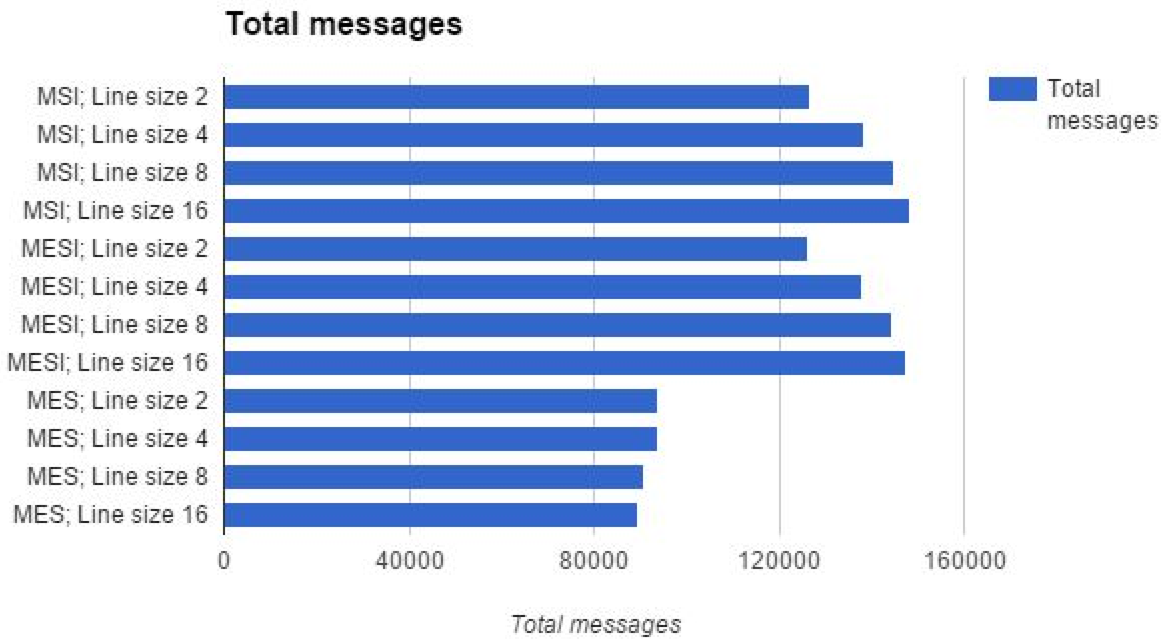

Data messages
MSI; Line size 2
MSI; Line size 4
MSI; Line size 8
MSI; Line size 16
MESI; Line size 2
MESI; Line size 4
MESI; Line size 8
MESI; Line size 16
MES; Line size 2
MES; Line size 4
MES; Line size 8
MES; Line size 16
0
10000
20000
30000
40000
Data messages
Data messages


Total messages
MSI; Line size 2
MSI; Line size 4
MSI; Line size 8
MSI; Line size 16
MESI; Line size 2
MESI; Line size 4
MESI; Line size 8
MESI; Line size 16
MES; Line size 2
MES; Line size 4
MES; Line size 8
MES; Line size 16
0
20000
40000
60000
80000
Total messages
Total messages

*Trace 2:*


Invalidation messages
Invalidati...
MSI; Line size 2
MSI; Line size 4
MSI; Line size 8
MSI; Line size 16
MESI; Line size 2
MESI; Line size 4
MESI; Line size 8
MESI; Line size 16
60000
64000
68000
72000
76000
Invalidation messages


Data messages
Data messages
MSI; Line size 2
MSI; Line size 4
MSI; Line size 8
MSI; Line size 16
MESI; Line size 2
MESI; Line size 4
MESI; Line size 8
MESI; Line size 16
MES; Line size 2
MES; Line size 4
MES; Line size 8
MES; Line size 16
0
25000
50000
75000
100000
Data messages

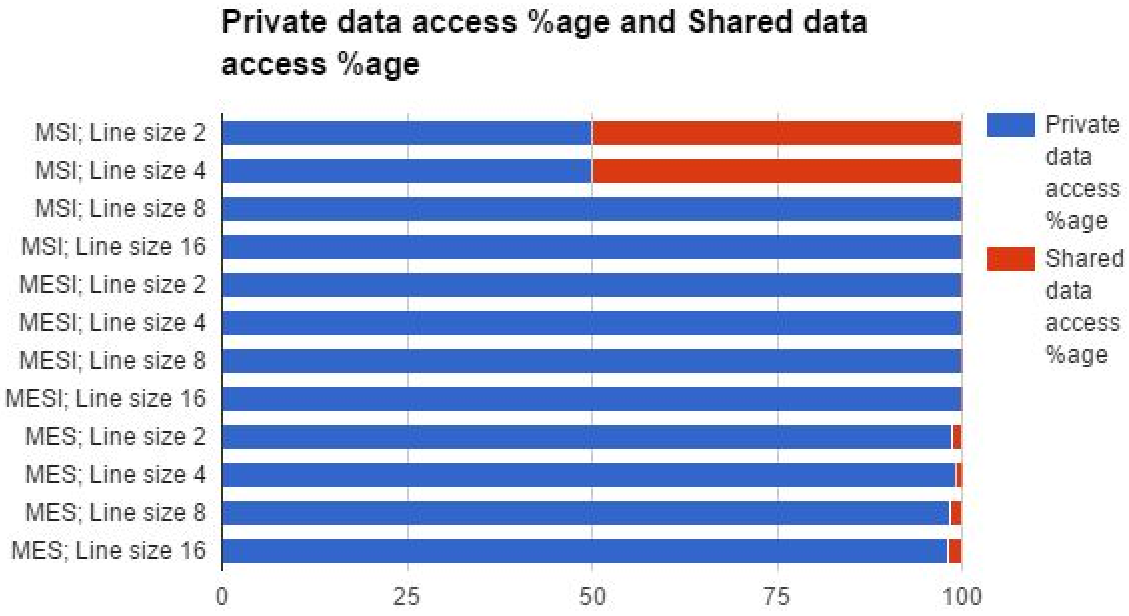


**Distribution of memory accesses:**

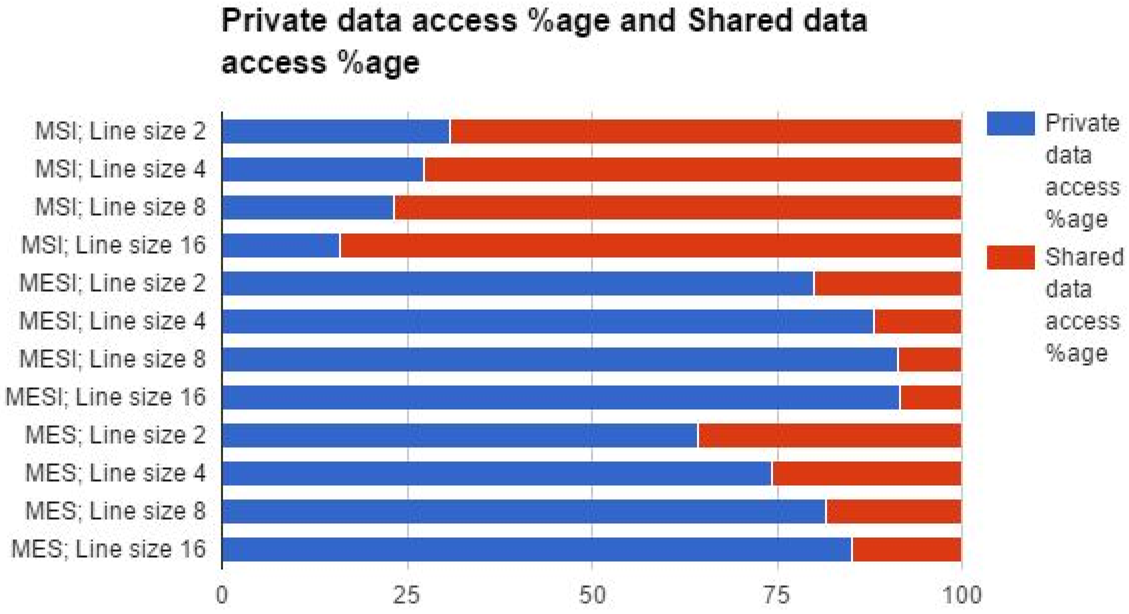
In trace 1, it would appear there is a breakpoint when moving from a line size of four to a line size of eight in the MSI protocol, where the ratio of accesses goes from roughly even between shared and private, to completely private. This could be due to the increase in cache misses, preventing shared data from being maintained in the cache due to eviction in favour of other addresses. In the MESI protocol, all accesses are to private storage. This could be due to the addition of the exclusive state, which comes under this category, and with the low hit-rate of the cache here there will be a low chance of a line ever transitioning from exclusive to shared state before being invalidated or evicted. The same general idea applies to the MES protocol, but here there are no invalidations, so a slightly higher chance (combined with better overall hit-rate) means some lines will transition to the shared state and remain there long enough to be accessed again.

In trace 2 the higher overall hit-rate allows for a greater percentage of shared accesses. The MSI protocol exhibits the highest number of shared accesses, due to any read producing a line in the shared state. More shared accesses are seen as the line size increases, likely due to shared lines being able to mutually coexist across several caches, making an access to them more probable. The MESI protocol has far more private accesses, due to the exclusive state's existence. As line size increases, the number of shared accesses decreases, likely due to the reduction in hit-rate and associated increase in eviction reducing the length of time a line will remain in the cache. This would then allow a higher chance for that line to be re-loaded in a private state after being evicted. In the MES protocol a similar situation occurs to that of the MESI protocol, minus the ability for lines to be invalidated. This lack of invalidation allows shared lines to remain in the cache longer, increasing the chance of shared line being accessed.

*Trace 1*



*Trace 2*



The set of results from the experiments, used to create the plots above, is available in the Appendix.

## Summary and Conclusion

Within this specific, simulated, context, it appears that the MES protocol is the best protocol of the three, exhibiting the highest hit-rate and lowest number of messages sent to the bus. The simple MSI protocol fared the worst overall, but was generally not too far behind the slightly more sophisticated MESI protocol.

However, in the more general context of practical computing the MES protocol may not be an ideal solution, as I could not find any reference to it beyond the handout provided. I am not clear on the exact reason for this, as the protocol is simple enough to be explained in a page of diagrams and notes. It would be utterly absurd to assume a protocol of this nature has never been thought of, so surely this cannot be the reason. Perhaps, in reality, the protocol creates so many implementation problems as to be impossible to utilize, or performs so poorly as to be completely useless, but even still one would expect to be able to find a report denouncing it as such.

A strange situation indeed, then, but not a unfavourable one: it allowed an interesting and seemingly unique take on cache protocols to be reasoned about and reported on, a thought-provoking exercise and look into the world of Parallel Architectures.

# Appendix

**Assumptions:**

- 16 bit addresses
- MES: "A data write message is also sent on the bus to update memory and requesting processor on a transition from M -> S." - this happens on a read OR a write
- MSI/MESI: remote write, causing local CPU to move from state M -> I, does not trigger a write-back.
- Whenever a cache line is loaded from memory, the full line is loaded. A full line is defined as enough addresses to fill a line, all with identical index and tag, with each possible offset represented.

**Trace 1 experiment results:**

| | MSI; Line size 2 | MSI; Line size 4 | MSI; Line size 8 | MSI; Line size 16 |
|---|---:|---:|---:|---:|
| Overall hit-rate %age | 0.53 | 0.26 | 0.07 | 0.03 |
| Invalidation messages | 32768 | 32678 | 32768 | 32768 |
| Data messages | 32756 | 32762 | 32764 | 32764 |
| Private data access %age | 50.05 | 50.1 | 100 | 100 |
| Shared data access %age | 49.95 | 49.9 | 0 | 0 |

| | MESI; Line size 2 | MESI; Line size 4 | MESI; Line size 8 | MESI; Line size 16 |
|---|---:|---:|---:|---:|
| Overall hit-rate %age | 0.52 | 0.26 | 0.07 | 0.03 |
| Invalidation messages | 32768 | 32768 | 32768 | 32768 |
| Data messages | 32756 | 32762 | 32764 | 32764 |
| Private data access %age | 100 | 100 | 100 | 100 |
| Shared data access %age | 0 | 0 | 0 | 0 |

| | MES; Line size 2 | MES; Line size 4 | MES; Line size 8 | MES; Line size 16 |
|---|---:|---:|---:|---:|
| Overall hit-rate %age | 0.78 | 0.65 | 0.59 | 0.55 |
| Invalidation messages | | | | |
| Data messages | 32801 | 32804 | 32827 | 32829 |
| Private data access %age | 98.64 | 99.06 | 98.44 | 98.07 |
| Shared data access %age | 1.36 | 0.94 | 1.56 | 1.93 |

**Trace 2 experiment results:**

| | MSI; Line size 2 | MSI; Line size 4 | MSI; Line size 8 | MSI; Line size 16 |
|---|---|---|---|---|
| Overall hit-rate %age | 7.67 | 4.75 | 3.06 | 2.41 |
| Invalidation messages | 63336 | 69279 | 72625 | 74590 |
| Data messages | 63211 | 69128 | 72236 | 73780 |
| Private data access %age | 30.7 | 27.41 | 23.26 | 15.94 |
| Shared data access %age | 69.3 | 72.59 | 76.74 | 84.06 |

| | MESI; Line size 2 | MESI; Line size 4 | MESI; Line size 8 | MESI; Line size 16 |
|---|---|---|---|---|
| Overall hit-rate %age | 7.67 | 4.75 | 3.06 | 2.14 |
| Invalidation messages | 62879 | 68805 | 72144 | 73810 |
| Data messages | 63211 | 69128 | 72236 | 73780 |
| Private data access %age | 80.08 | 88.17 | 91.27 | 91.62 |
| Shared data access %age | 19.92 | 11.83 | 8.73 | 3.83 |

| | MES; Line size 2 | MES; Line size 4 | MES; Line size 8 | MES; Line size 16 |
|---|---|---|---|---|
| Overall hit-rate %age | 11.93 | 9.54 | 7.23 | 5.5 |
| Invalidation messages | | | | |
| Data messages | 94024 | 93974 | 90786 | 89495 |
| Private data access %age | 64.22 | 74.44 | 81.61 | 85.1 |
| Shared data access %age | 35.78 | 25.56 | 18.39 | 14.9 |