1 | Simulator Design

The simulator in question is written in Python 3.4 and can be executed with the following command (when simulator.py is set to executable):

```
./simulator.py -f <filename> -p <protocol> -b <block_size>
```

or without setting the file as executable with

```
python3.4 simulator.py -f <filename> -p <protocol> -b <block_size>
```

Where:

- `filename` indicates the trace filename to use
- `protocol` can be one of `[msi, mesi, mes]`
- `block_size` expects an integer, one of `[2, 4, 8, 16]` and determines the cache block size

Alternatively, omitting the filename the program expects an input stream with instructions and commands. We can 'pipe' input from other files into the program this way or use the simulator interactively.

The program supports the required commands such as `v, p, h, i`. When the filename argument is used, the filename will need to contain the commands as the user will not be prompted for command entry.

The program is split across multiple files, namely `simulator.py, models.py and protocols.py. simulator.py` is the main entry point of the program responsible for parsing command line arguments, reading the input stream and processing each instruction from the input stream. Actual underlying state and behavior is delegated to `models.py` which implements the DirectMapped cache, the cache bus as well as other encapsulations. The `protocols.py` file is responsible for implementation of the FSM which the coherence protocols rely on.

The program aims to mimic the structure inside real CPU architectures by aligning each individual cache to a CPU with all of the connected to a bus responsible for message passing.

Firstly, an instruction is parsed from the input stream and it is directed to its corresponding cache. The cache attempts to retrieve the cache line, validates the tag and the state and if successful a hit occurs. In the case of a miss, the cache responds with failure.

Secondly, given the current state (invalid if cache access was a miss) of the cache line and the action (read-hit, read-miss, write-hit, write-miss) requests local state transition from the protocol. The local cache is then updated with the new state. If the instruction is a write and remote caches store a modified entry, invalidate broadcast is issued on the bus.

Thirdly, a message is placed on the bus. The message is responsible for updating the state of the remote caches by determining the appropriate transition and setting the states accordingly. In the case of a miss in the remote caches, no action is taken (except for MES), otherwise the state is updated.

Finally, statistics on individual actions and state transitions are collected. The overall statistics in a crude form are printed at the end of execution. The user can utilize the required commands to display statistics in a more digestible form.

2 | Sample Traces
2.1 | MSI

```
v
P0 R 1       # Read Miss: I -> S
P0 R 1       # Read Hit: S -> S
P1 R 1       # Read Miss: I -> S
P0 W 1       #  Write  Hit  (upgrade):  S  ->  M,  Triggers  invalidate  on  P1  (considered
miss)
P0 W 1       # Write Hit: M -> M
P0 R 1       # Read Hit: M -> M
P1 W 1       # Write Miss: I -> M, Triggers invalidate on P0,
P1 W 20      # Write Miss: I -> S

# 5 misses, 3 hits => 3 / 8 hit rate = 37.5%
h

# P0: {1: I}
# P1: {1: M, 20: S}
p

# 2 invalidate broadcasts, 2 lines invalidated
i
```

This trace, available as *msi.trace.1.txt* file in the project directory, showcases the state transitions on 2 processors. Additionally, it displays the usage of the flags to display additional information. Furthermore, it showcases the scenario where an invalidate is triggered from the Shared state to the Modified state which counts as a miss. The output of the file is shown below:

```
$ python3.4 simulator.py -f msi.trace.txt -p msi -b 4
Arguments used: {'block_size': 4, 'file': 'msi.trace.1.txt', 'protocol': 'msi'}
Running MSI cache coherence simulator on Direct Mapped cache with block size of 4 words
on 4 CPUs
Verbose switched to: True
A Read miss by P0 to word 1 looked for tag 0 in block 0, found in state Invalid. No
other copies.
A Read hit by P0 to word 1 looked for tag 0 in block 0, found in state Shared. No other
copies.
A Read miss by P1 to word 1 looked for tag 0 in block 0, found in state Invalid. Other
CPUs are in states ['P0 - Shared']
A Write hit by P0 to word 1 looked for tag 0 in block 0, found in state Shared. No
other copies.
A Write hit by P0 to word 1 looked for tag 0 in block 0, found in state Modified. No
other copies.
A Read hit by P0 to word 1 looked for tag 0 in block 0, found in state Modified. No
other copies.
A Write miss by P1 to word 1 looked for tag 0 in block 0, found in state Invalid. Other
CPUs are in states ['P0 - Modified']
A Write miss by P1 to word 20 looked for tag 0 in block 5, found in state Invalid. No
other copies.
Hit Rate: 37.50%. Private hits: 66.67%. Shared hits: 33.33%
P0
  0: 0 (Invalid)
P1
  0: 0 (Modified)
  5: 0 (Modified)
P2
P3
Invalidation broadcasts: 2. Lines invalidated: 2
```

2.2 | MESI

```
v
P0 R 1      # Read miss & no sharing: I -> E
P0 R 1      # Read hit: E -> E
P1 R 1      # Read miss & sharing: I -> S, triggers Remote read miss on P0 (E -> S)
P0 R 50     # Read miss & no sharing: I -> E
P0 W 50     # Write Hit (silent): E -> M
P2 R 50     # Read miss & sharing: I -> S, triggers remote read miss on P0 (M -> I)
P0 R 50     # Read miss & sharing: I -> S
P2 W 50     # Write hit: S -> M, invalidate broadcast to P0

# 3 hits, 5 misses => 3/8 hit rate => 37.5%
# 2 private hits, 1 shared
h

# 1 invalidate sent, 1 line invalidated
i

# P0 {1: S, 50: I}
# P1 {1: S}
# P2 {50: M}
p
```

In this trace, *mesi.trace.txt*, we focus on the correct transitions between the Exclusive state, silent and invalidating writes and the resulting hit rate as well as invalidations. The output of the trace is the following:

```
python3.4 simulator.py -f mesi.trace.txt -p mesi
Arguments used: {'protocol': 'mesi', 'block_size': 4, 'file': 'mesi.trace.txt'}
Running MESI cache coherence simulator on Direct Mapped cache with block size of 4
words on 4 CPUs
Verbose switched to: True
A Read miss by P0 to word 1 looked for tag 0 in block 0, found in state Invalid. No
other copies.
A Read hit by P0 to word 1 looked for tag 0 in block 0, found in state Exclusive. No
other copies.
A Read miss by P1 to word 1 looked for tag 0 in block 0, found in state Invalid. Other
CPUs are in states ['P0 - Exclusive']
A Read miss by P0 to word 50 looked for tag 0 in block 12, found in state Invalid. No
other copies.
A Write hit by P0 to word 50 looked for tag 0 in block 12, found in state Exclusive. No
other copies.
A Read miss by P2 to word 50 looked for tag 0 in block 12, found in state Invalid.
Other CPUs are in states ['P0 - Modified']
A Read hit by P0 to word 50 looked for tag 0 in block 12, found in state Shared. Other
CPUs are in states ['P2 - Shared']
A Write hit by P2 to word 50 looked for tag 0 in block 12, found in state Shared. No
other copies.
Hit Rate: 37.50%. Private hits: 66.67%. Shared hits: 33.33%
Invalidation broadcasts: 1. Lines invalidated: 1
P0
  0: 0 (Shared)
  12: 0 (Invalid)
P1
  0: 0 (Shared)
P2
  12: 0 (Modified)
P3
```

2.3 | MES

```
v
P0 R 10     # Read miss: E
P0 R 10     # Read hit: E -> E
P0 W 10     # Write hit: E -> M
P1 R 10     # Read miss & shared: S. Causes remote miss on P0 (M -> S)
P1 W 10     # Write hit & send update: S -> S. P0 receives write update (S -> S)
P2 W 10     # Write miss & shared & send update: S. P0, P1 receive write update (S ->
S)
P3 W 20     # Write miss & not shared: M
P0 R 10     # Read hit: S -> S

# 4 misses, 4 hits => 0.5% hit rate
# 2 private hits, 2 shared hits
# 2 write updates
h

# P0 {10: S}
# P1 {10: S}
# P2 {10: S}
# P3 {20: M}
p
```

The above trace, *mes.trace*, produces the following output which matches the expectation:

```
python3.4 simulator.py -f mes.trace.txt -p mes
Arguments used: {'file': 'mes.trace.txt', 'protocol': 'mes', 'block_size': 4}
Running MES cache coherence simulator on Direct Mapped cache with block size of 4 words
on 4 CPUs
Verbose switched to: True
A Read miss by P0 to word 10 looked for tag 0 in block 2, found in state None. No other
copies.
A Read hit by P0 to word 10 looked for tag 0 in block 2, found in state Exclusive. No
other copies.
A Write hit by P0 to word 10 looked for tag 0 in block 2, found in state Exclusive. No
other copies.
A Read miss by P1 to word 10 looked for tag 0 in block 2, found in state None. Other
CPUs are in states ['P0 - Modified']
A Write hit by P1 to word 10 looked for tag 0 in block 2, found in state Shared. Other
CPUs are in states ['P0 - Shared']
A Write miss by P2 to word 10 looked for tag 0 in block 2, found in state None. Other
CPUs are in states ['P0 - Shared', 'P1 - Shared']
A Write miss by P3 to word 20 looked for tag 0 in block 5, found in state None. No
other copies.
A Read hit by P0 to word 10 looked for tag 0 in block 2, found in state Shared. Other
CPUs are in states ['P1 - Shared', 'P2 - Shared']
Hit Rate: 50.00%. Private hits: 50.00%. Shared hits: 50.00%
P0
  2: 0 (Shared)
P1
  2: 0 (Shared)
P2
  2: 0 (Shared)
P3
  5: 0 (Modified)
```

3 | Results

From brief analysis of the trace contents, trace1.txt appears to be dealing mostly with memory accesses localized to each CPU with a relatively small portion of overlapping (between CPUs) memory accesses. This could correspond to a partitioned workload where each CPU is responsible for its own portion of work without high levels of cooperation between the CPUs.

On the other hand, trace2.txt deals with a high level address sharing across the CPUs. A relatively small portion of memory accesses are localized throughout the whole trace. This workload could correspond to an application with high levels of cooperation between individual CPUs.

3.1 | MSI

trace1.txt						
Block Size	*Cache Lines*	*Hit Rate*	*Private Hits*	*Shared Hits*	*Invalidate broadcasts*	*Lines invalidated*
2	1024	82.54%	59.47%	40.53%	16640	99
4	512	91.27%	72.18%	27.82%	8328	51
8	256	95.63%	77.73%	22.27%	4168	24
16	128	97.81%	80.33%	19.67%	2088	12

Firstly, from the results we can observe that as the block size increases, the hit rate increases. Additionally, so does the number of private hits proportion which leads to a decrease in shared hits.

Secondly, an increase in block size leads to a decrease in the total number of invalidate broadcasts as well as the total number of lines invalidated. This can be explained through workload address locality.

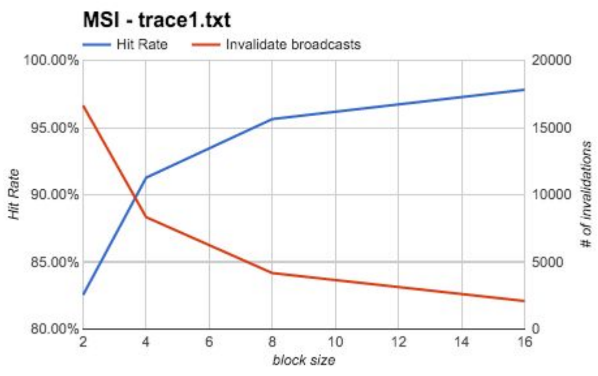
Thirdly, we can observe that the hit rate is directly correlated to the number of invalidate broadcasts. The lower the number of invalidate broadcasts, the higher the overall hit rate. This is reasonable as the primary concern of a cache coherence protocol is to minimize the overall miss rate which can be achieved through minimizing the total number of invalidations.

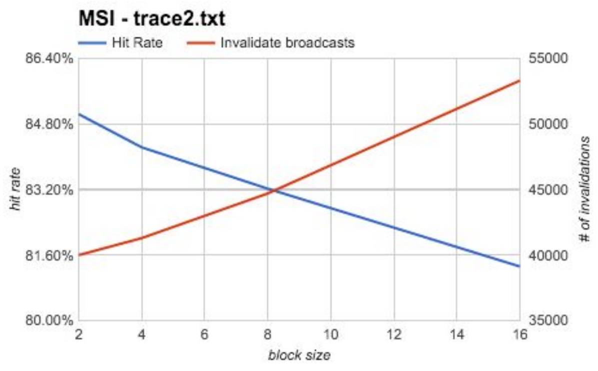
trace2.txt						
Block Size	*Cache Lines*	*Hit Rate*	*Private Hits*	*Shared Hits*	*Invalidate broadcasts*	*Lines invalidated*
2	1024	85.04%	8.34%	91.66%	39993	47124
4	512	84.23%	8.24%	91.76%	41290	52559
8	256	83.22%	7.77%	92.23%	44697	61300
16	128	81.32%	6.81%	93.19%	53311	78733

In trace2.txt, we can observe that the inverse relationship holds. As the block size is increased, the hit rate decreases. This is also true for the number of private hits, as block size increases (decrease total number of blocks), the number of hits which are private to the CPU decreases.

Similarly, an increase in the block size leads to an increase the total number of invalidate broadcasts and corresponding line invalidations.

Plotting the block size against the hit rate and the number of invalidate broadcasts, we can observe the relationship between hit rate and broadcasts in each of the traces.





From the figure above, we can see the inverse relationship of the two traces. In trace1.txt, an increase in block size also improves hit rate, however, with diminishing returns as the size grows larger. This can be observed by the hit rate flattening out. Similarly, a further increase in the block size achieves disproportionately lower decrease in the total number of invalidations.

In the case of trace2.txt, the rate of decrease in hit rate as block size increases is linear, that is, doubling the cache size leads roughly to a proportionate decrease in hit rate which also proportionately increases the number of invalidate broadcasts.

Trace1.txt shows better performance due to the locality of the trace itself. The converse is true for trace2.txt as it deals with a high level of address space sharing.

3.2 | MESI

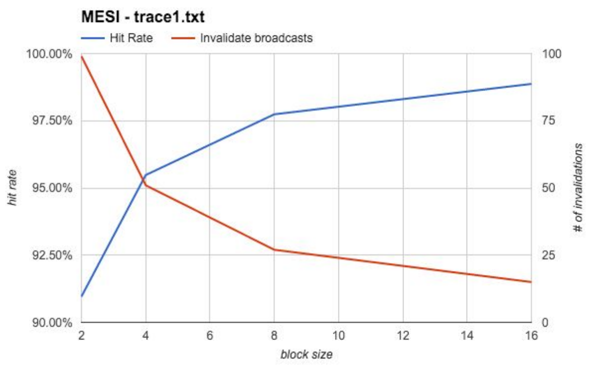
trace1.txt						
Block Size	*Cache Lines*	*Hit Rate*	*Private Hits*	*Shared Hits*	*Invalidate broadcasts*	*Lines invalidated*
2	1024	90.96%	99.48%	0.16%	99	99
4	512	95.48%	99.87%	0.13%	51	51
8	256	97.74%	99.88%	0.12%	27	24
16	128	98.87%	99.88%	0.12%	15	12

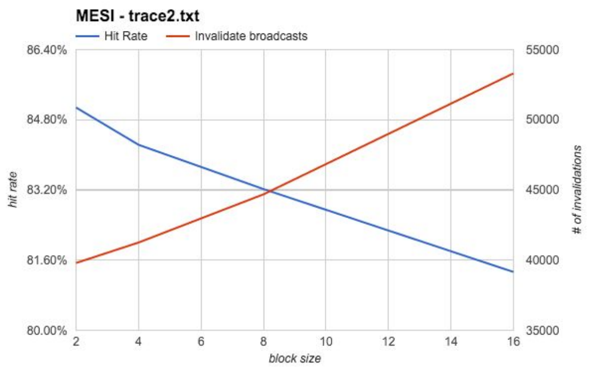
In the case of trace1.txt, as block size increases, so does the hit rate. Additionally, the number of invalidate broadcasts decreases. As the hit rate increases with block size, so does the number of private hits. We can also observe that the relative gains obtain by increasing the block size have diminishing returns as the block size is increased further, that is, an increase in block size leads to a disproportional increase in hit rate.

Trace2.txt, on the other hand, observes the opposite relationship. An increase in block size leads to a decrease in hit rate. In fact, an increase in block size leads to disproportionately larger decrease in hit rate. The hit rate is decreased as the number of invalidates increases with block size. This is due to a large number of shared addresses in the workload requiring invalidations.

trace2.txt						
Block Size	*Cache Lines*	*Hit Rate*	*Private Hits*	*Shared Hits*	*Invalidate broadcasts*	*Lines invalidated*
2	1024	85.08%	8.45%	91.55%	39795	47124
4	512	84.23%	8.27%	91.73%	41253	52559
8	256	83.22%	7.79%	92.21%	44688	61300
16	128	81.33%	6.82%	93.18%	53306	78733

Plotting the block size against the hit rate and the number of invalidate broadcasts, we can observe the relative impact of increasing the block size.





From the graphs above, we can observe for trace1.txt, the relative gain in hit rate as block size increases is decreasing, leading to diminishing returns. Similarly, the number of invalidates decreases at a slower pace as the block size is increased further.

On the other hand, in the case of trace2.txt, an increase in block size leads to a decrease in hit rate and an increase in the number of invalidate broadcasts. The effect is near linearly, meaning that we achieve a relatively proportional decrease in hit rate for each increase in block size.

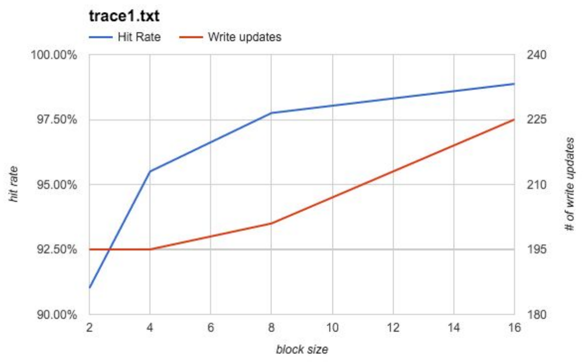
3.3 | MES

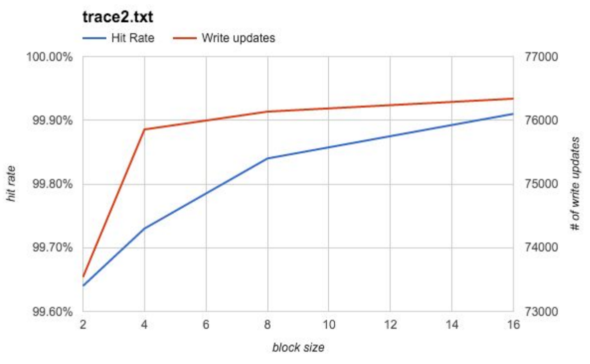
trace1.txt						
Block Size	*Cache Lines*	*Hit Rate*	*Private Hits*	*Shared Hits*	*Write updates*	*Lines write updated*
2	1024	91.01%	99.46%	0.54%	195	198
4	512	95.5%	99.43%	0.57%	195	99
8	256	97.75%	99.4%	0.6%	201	51
16	128	98.87%	99.32%	0.68%	225	27

As the number of blocks increases, the hit rate improves also. The number of private hits decreases due to the protocol updating the values and moving them into the shared state. The number of write updates increases with the block size as the total size of the cache decreases. The number of actual lines updated, however, decreases with block size.

trace2.txt						
Block Size	*Cache Lines*	*Hit Rate*	*Private Hits*	*Shared Hits*	*Write updates*	*Lines write updated*
2	1024	99.64%	0.92%	99.08%	73539	838
4	512	99.73%	0.13%	99.87%	75855	506
8	256	99.84%	0.06%	99.94%	76135	256
16	128	99.91%	0.02%	99.98%	76338	128

In the case of trace2.txt, the hit rate increases as block size is increased. The number of private hits decreases while the number of shared hits increases due to the protocol's design. The number of write updates increases slightly as block size is increased but remains relatively high throughout. The number of actual lines updated decreases as the block size increases.





From the relationship between hit rate and write updates shown above, we can see that there is a positive correlation between the hit rate and the number of write updates. This is the case for both trace1 and trace2. In the case of trace1, the hit rate flattens out faster than the number of write updates while for trace2, the converse is the case, that is the hit rate keeps improving more steadily while the number of write updates flattens out after block size of 8.

4 | Analysis
Three distinct coherence protocols have been explored, MSI and MESI which employ the invalidation policy while the MES protocol which updates other caches with values on updates.

4.1 | Hit rate
Out of all the protocols, the MES protocol performed the best in terms of hit rate, achieving up to 98.87 and 99.91 percent hit rate in their best cases on trace1 and trace2 respectively. The second best protocol observed is the MESI protocol achieving 98.87 and 81.33 percent hit rate respectively.

The hit rate of a cache is good indicator if it's performance, however, in a cache coherence protocol where caches need to be kept synchronized the number of messages and their respective cost can be an indicator of their relative performance. Higher number of messages puts additional stress on the hardware and drives cost requirements.

4.2 | Messages

If we compare the caches in terms of the number of messages sent on the bus, then the MESI protocol outperforms both MES and MSI. This is due to the ability to retain local addresses in an Exclusive state without the requirement to notify other caches about the state leading to a lower number of messages sent. The MESI protocol has sent 15 and 53306 messages on the bus respectively for the two trace files in the best performing case.

4.3 | Traces

Trace1.txt focuses heavily on localized workloads and therefore generates significantly less messages in all of the protocols. Trace2.txt incurres a large number of shared addresses across CPUs resulting in a much higher number of messages broadcasted. The relative merits of each protocol therefore also depend on the intended use of the architecture.

4.3 | Scalability

The MES protocol appears to be the most scalable when hit rate vs the number of messages is considered, however, the actual cost updating the cache frequently may be significant.