Parallel Architectures, Assignment 3
Ewan Leaver, s0800696

All aspects of the coursework were implemented and completed. The assignment instructions in regards to latencies were interpreted to produce the following values for a given access, and are produced as such by the simulator:

Reads Accesses:
- Read Bypass - 1 cycle
- Found in local cache - 2 cycles
- Found in other cache (shared or modified) - 22 cycles (cache access + bus transaction)
- Memory access - 222 cycles (cache, bus and memory latencies)

Write Accesses:
- Modified locally - 202 cycles (No need for bus transactions)
- Modified elsewhere - 222 cycles (Bus transaction to invalidate)
- Shared elsewhere - 222 cycles (Bus transactions to invalidate, assumed in parallel)
- Invalid - 222 cycles (Bus transactions to check other caches)

It was assumed that bus transactions occurred in parallel if multiple caches needed to be invalidated, or for similar operations.

All parts of the assignment were attempted and implemented. The simulator is written in C++, and uses classes to model the four processors.

The simulator is initialised using the file build.sh (see the accompanying readme.txt). Within this file, the input trace file, number of lines, and line size are all specified as arguments, and can be changed if required. Upon execution of the build script, the main simulator program is called, which in turn instantiates the four processor objects. The main simulator class acts as the MSI snooping protocol, and maintains all global state information and functionality. After initial start-up preparations, readTrace() is called, which begins examining the trace file line-by-line, and sends each memory operation to the relevant processor.

Cache lines are also modelled as a class, and are by the processors to store the local MSI state for each line. The cache is constructed through a vector of such lines, and is resized to match the number of lines in the cache upon processor instantiation.

The cache line class also has multiple other information fields, which are not used by the processor classes, but are used by the main simulator class instead. The simulator maintains a vector of these lines (up to 20000, currently), and uses them to track global MSI states for each line, as well as tracking various qualities for calculating the end simulator statistics. Upon beginning another memory access from the trace file, the processor will calculate the corresponding line number (based on the input line size), and adjust the relevant fields before initiating a load or store operation at the relevant processor. Each line object also stores the line size, which is used as a parameter for its instantiation.

Within the processor class, the majority of the functionality pertains to the load and store functions. The processor is initialised by providing a name, line number and line size as argumentsThese simply take an address value as an integer, and proceed to calculate the line offset, index and tag values. The processor actually maintains two sets of cache information - the first is a 2D vector array of doubles, which stores the current occupying memory location in every word of used cache lines. For example, the a read to address 7, and a write to address 512 would result in the following cache arrangement (assuming a line size of 4, and 128 lines in the cache).

```
line 0: [512] [513] [514] [515] // Addr 512 - index = 0

line 1: [ 5 ] [ 6 ] [ 7 ] [ 8 ]
line2:  [   ] [   ] [   ] [   ] // Lineunused
```

The previously mentioned vector of line objects is also maintained, and will record the local MSI information for each line. A line's state field can have one of the following four values:

-2 = unusued
-1 = used, but has been invalidated
0 = shared
1 = modified

By testing if a required line's state is used but invalid, a coherence miss can be detected, and the relevant counter incremented. After this check, it is further tested whether the line is currently NOT modified locally. If this check succeeds, the line state is changed to shared – if a line is currently locally modified, this state would remain. Finally, if the requested address's value is found in the cache, a hit is recorded, otherwise a miss is noted. In both cases, the relevant counters are incremented.

The processor also has a method to force change the state of the line of a given address, required to model invalidations in the local caches. As well as performing this simple operation, the method also increments the local invalidations counter, if the invalid state is being enforced.

This concludes the explanation of the simple inner workings of the processor and line objects, and now we will briefly explain the functionality of the main class file, which acts as the MSI snooping protocol, and enforces the necessary invalidations.

Upon a trace line read, the corresponding line number and offset of the address is calculated.

As well as tracking the global state of each line, other

If the access is a read, the simulator first checks whether the corresponding line is modified. If it is, the processor with modifying ownership is made to invalidate the line in it's cache, and the line's 'owner' is set to an invalid value. Following this, the line's state is set to shared, the requesting processor is flagged as sharing within the line object, and the processor's load operation is called.

For write accesses, the process is a little more complex.

- If the line's current state is 0 (shared – see above for state values), the simulator checks whether each of the other processors is flagged as sharing the line. If any are, they are simply told to invalidate the line in their local caches. The line's state is then recorded as by modified by the requesting processor, and the processor's store function is called.

- The code for an unused or invalid code line shares the second part of this code, simply setting the line's state as modified, and calling the store function of the relevant processor.

- If the state is modified, it is first checked to see if the line’s modifying processor IS the currently requesting processor. If so, no state enforcement is required, and the processor’s store function is immediately called. Otherwise, the line is invalidated in the cache of the processor marked as modifying the line, the requesting processor is marked as the new modified, and the processor’s store function is called on the address.

This concludes the global state enforcement code within the main class.

Section 1 - Verification of Simulator

Note about the terminal output:
Lines beginning with 'P0 ...' or equivalent are lines printed locally from each processor, and indicate the completion of a given access. Any other lines are produced by the main class file, in order to help highlight the internal processes for the sake of validating the simulator.

Executing the following trace, as mentioned in the assignment, using a retire-at-2 write policy:

```
P0 W 2
P0 R 6
P0 W 45
P0 W 26
P0 R 26
```

Produces the following output:

```
PUSHING ADDR 2 ONTO WRITEBUFFER FOR PROC P0
P0   R   Addr:      6  Tag:    0  Index:    1  Offset:    2  Res:  miss   Latency:   222
PUSHING ADDR 45 ONTO WRITEBUFFER FOR PROC P0
***     STARTING WRITE OF ADDR 2 FOR P0.    Current Latency: 222,    Completed by: 444
PUSHING ADDR 26 ONTO WRITEBUFFER FOR PROC P0
P0   R   Addr:     26  ***** READ BYPASS *****                        Latency:   223
***     COMPLETED WRITE OF ADDR 2 FOR P0. Latency: 444
P0   W   Addr:     45  Tag:    0  Index:   11  Offset:    1  Res:  miss   Latency:   666
P0   W   Addr:     26  Tag:    0  Index:    6  Offset:    2  Res:  miss   Latency:   888
```

```
- The initial write to address 2 is pushed onto the writebuffer.
- Read of 6 misses, and cache, bus and memory latencies are accumulated.
- Address 45 is pushed onto the writebuffer.
- Reached two elements in buffer, first write to address 2 commences
- Meanwhile address 26 is added to the writebuffer. The write to 45 must wait, however.
- The final read to 26 is a read bypass, resulting in a latency of 1
- The write to address 2 completes at a latency of 444. As the trace has finished, the
  remaining writes are flushed from the writebuffer.
-
```

Demonstration of flushing the write-buffer

Using the following trace, with retire-at-2 and a writebuffer length of 4:

```
P0 W 2
P0 R 2
P0 W 3
P0 W 12
P0 W 1
P0 W 5
P0 R 128
```

```
PUSHING ADDR 2 ONTO WRITEBUFFER FOR PROC P0
P0   R   Addr:      2  ***** READ BYPASS *****                        Latency:   1
PUSHING ADDR 3 ONTO WRITEBUFFER FOR PROC P0
***     STARTING WRITE OF ADDR 2 FOR P0.    Current Latency: 1,    Completed by: 223
PUSHING ADDR 12 ONTO WRITEBUFFER FOR PROC P0
PUSHING ADDR 1 ONTO WRITEBUFFER FOR PROC P0
***     FLUSHING WRITEBUFFER FOR P0
P0   W   Addr:      2  Tag:    0  Index:    0  Offset:    2  Res:  miss   Latency:   223
P0   W   Addr:      3  Tag:    0  Index:    0  Offset:    3  Res:  hit   Latency:   425
P0   W   Addr:     12  Tag:    0  Index:    3  Offset:    0  Res:  miss   Latency:   647
P0   W   Addr:      1  Tag:    0  Index:    0  Offset:    1  Res:  hit   Latency:   849
PUSHING ADDR 5 ONTO WRITEBUFFER FOR PROC P0
P0   R   Addr:    128  Tag:    0  Index:   32  Offset:    0  Res:  miss   Latency:   1071
P0   W   Addr:      5  Tag:    0  Index:    1  Offset:    1  Res:  miss   Latency:   1293
```

```
- P0 W 2 pushed to buffer
- P0 R 2 completes - read bypass, so latency = 1 cycle
- P0 W 3 pushed to buffer, P0 W 2 starts.
- the next writes fill the buffer, prompting it to be flushed. Some accesses are to
  locally modified lines (latency of 202). Otherwise, latency of 222 cycles.
```

Reading and Writing to shared & modified lines

Executing the following trace, with retire-at-1 and an infinite write buffer:

```
P0 R 2
P1 R 2
P0 W 3
P0 W 0
P0 R 12
P3 W 1
P0 W 0
```

Provides the following output:

```
P0   R   Addr:      2  Tag:    0  Index:    0  Offset:    2  Res:  miss   Latency:   222
P1   R   Addr:      2  Tag:    0  Index:    0  Offset:    2  Res:  miss   Latency:   22
PUSHING ADDR 3 ONTO WRITEBUFFER FOR PROC P0
***     STARTING WRITE OF ADDR 3 FOR P0.    Current Latency: 222,    Completed by: 444
PUSHING ADDR 0 ONTO WRITEBUFFER FOR PROC P0
P0   R   Addr:     12  Tag:    0  Index:    3  Offset:    0  Res:  miss   Latency:   444
PUSHING ADDR 1 ONTO WRITEBUFFER FOR PROC P3
***     STARTING WRITE OF ADDR 1 FOR P3.    Current Latency: 0,    Completed by: 222
PUSHING ADDR 0 ONTO WRITEBUFFER FOR PROC P0
***     COMPLETED WRITE OF ADDR 3 FOR P0. Latency: 444
P0   W   Addr:      3  Tag:    0  Index:    0  Offset:    3  Res:  miss   Latency:   444
***     COMPLETED WRITE OF ADDR 1 FOR P3. Latency: 222
P3   W   Addr:      1  Tag:    0  Index:    0  Offset:    1  Res:  miss   Latency:   222
P0   W   Addr:      0  Tag:    0  Index:    0  Offset:    0  Res:  hit   Latency:   666
P0   W   Addr:      0  Tag:    0  Index:    0  Offset:    0  Res:  hit   Latency:   868
```

```
- The read at P0 for address 2 completes. Has cache, bus and mem latencies
- Address 2 is in the cache of P0, so P1 only accumulates cache and bus latencies
- P0's write to 3 is added to the buffer and starts immediately due to retire-at-1
- Write to 0 is also added to the buffer, but must wait for the previous write to
  complete
- Read to 12 completes, needs cache, bud, and mem latencies
- Write by P3 to address 1 is added to the buffer and commences.
- Final write by P0 to 0 added to buffer. Still has to wait.
- P3's write to 1 completes, was modified by P0, so needed cache, bus and mem latencies
  to invalidate
- P0's two writes to 0 complete. The first needs to invalidate the line in P3's cache,
  so needs cache + bus + mem. The second has the line in P0's cache, so only cache + mem
  latencies required.
```

Section 2 – Design and Execution of Experiments

Question i: Variation of Write-Buffer Size

Table 1 – trace1.out

	Max Latency (cycles)	P0 Read Miss Rate	P0 Write Miss Rate	P0 Total Miss Rate
SC	2144300	0.0539551	1	0.211629
TSO, Buffer Length = 4	2019540	0.0539551	0.437256	0.117839
TSO, Buffer Length = 8	1836060	0.0539551	0.33252	0.100382
TSO, Buffer Length = 16	1826690	0.0539551	0.294922	0.0941162
TSO, Buffer Length = 32	1772980	0.0539551	0.275391	0.090861
TSO, Buffer Length = 64	1762920	0.0539551	0.269043	0.0898031

Table 2 – trace2.out

	Max Latency (cycles)	P0 Read Miss Rate	P0 Write Miss Rate	P0 Total Miss Rate
SC	4974100	0.140183	0.714679	0.209744
TSO, Buffer Length = 4	4947770	0.139841	0.713166	0.20926
TSO, Buffer Length = 8	4830260	0.139461	0.707275	0.208213
TSO, Buffer Length = 16	4644370	0.137711	0.693114	0.204961
TSO, Buffer Length = 32	4564870	0.133683	0.657713	0.197134
TSO, Buffer Length = 64	4510030	0.127116	0.59145	0.183338

Question ii: Variation of Retire Policy

Table 3 – trace1.out

	Max Latency (cycles)	P0 Read Miss Rate	P0 Write Miss Rate	P0 Total Miss Rate
SC	2144300	0.0539551	1	0.211629
TSO, retire-at-1	1772980	0.0539551	0.275391	0.090861
TSO, retire-at-2	1774420	0.0539551	0.275024	0.0908
TSO, retire-at-4	1781990	0.0539551	0.271362	0.0901896
TSO, retire-at-8	1832480	0.0539551	0.276978	0.0911255
TSO, retire-at-16	1881400	0.0539551	0.276367	0.0910238

Table 4 – trace2.out

	Max Latency (cycles)	P0 Read Miss Rate	P0 Write Miss Rate	P0 Total Miss Rate
SC	4974100	0.140183	0.714679	0.209744
TSO, retire-at-1	4564870	0.133683	0.657713	0.197134
TSO, retire-at-2	4577090	0.133906	0.659172	0.197507
TSO, retire-at-4	4605980	0.133854	0.661712	0.197768
TSO, retire-at-8	4664370	0.133698	0.656632	0.197016
TSO, retire-at-16	4774380	0.133526	0.654037	0.196551

Section 3 - Report

This report references the results given in the tables of Section 2.
Due to the vast quantity of data made available from the simulator results, for all statistics only the data produced by P0 has been presented in this document. This choice was made because the statistics for each processor are roughly equivalent for each experiment. If differences were noticed between processor statistics, they are mentioned in the report.

Table 1 illustrates the variation of performance of P0 when altering the write buffer size for trace1.out. It can be clearly seen that the read miss rate remains constant. Due to the nature of trace1, this is likely due to the pattern of reads and writes (ie of several successive reads from different addresses, followed by a write to the first address). The 2nd assignment established that trace1 represented a program cycling over arrays of input data and performing arithmetic operations, before writing back into the first address.

It would appear to be the case that any read accesses which cause a hit are located in close proximity to the original instruction which loaded it into the cache. Generally, the read instructions access addresses previously accessed by recent read instructions, and there appears to be no shared data between processes. This means that a relaxation of the ordering between reads and writes would have no influence on the read miss rate. However some reads access the same address as a recent write, resulting in read bypasses, and contributing to a lowered latency. Due to the locality of these accesses, the number of read bypasses remains constant when increasing the writebuffer size, and so read accesses do not further decrease the latency. Further decreases in latency are likely to be due to removing the writes from the critical path.

It is obvious however, that increasing the write buffer size causes a decrease in write miss rates. This could be due to a couple of factors. It may be that by storing writes in a buffer, accesses from another processor which would have previously invalidated the line will now access the line first. When the processor finally writes to the line, there is therefore a reduced chance of the line being invalidated. There is an initial sudden decrease in the write miss rate for trace1 when TSO is introduced, even with a retire-at-1 policy.

It may also be due to the decoupling of the write order between processors, ie, before where two or more processors attempted to write to the same address multiple times, it may have caused one or more invalidations for both processors. If if order is decoupled, each processor’s access to the address is more likely to be further removed from the other processor, reducing the number of invalidations and decreasing the miss rate.

It may also be down to read bypasses. If a process’s write gets added to the buffer, any reads to the same address will hit upon checking the buffer. This means that regardless of whether another processor owns the line or not, the read access will not attempt to invalidate the line. If another processor does own the line, they can safely continue to modified it until the write from the other processor is completed, resulting in improved miss rates.

This is also likely occurring within trace2.out. However, it is worth noting that in trace 2, only P0 has a noticeable number of read bypasses, and therefore is the only processor for which the read miss rate changes significantly. For all other processors, the read rate remains largely constant while varying the write buffer size. However, in all processors the hit miss rate decreased in a similar fashion. This is likely down to the read and write pattern for each processor; as assignment 2 established trace2 as having a lot of coherence communication, perhaps typical of a multithreaded program. This perhaps indicates that P0 is executing the main thread, with all other processes constituting workers communicating with it.

The results gained from question two were unexpected however, as there appears to be little to be gained from varying the value of retire-at-N. The constant read miss rate of trace1 is explained in much the same way as for question 1. It appears that the structure of this program is such that TSO cannot improve the read miss rate. There is very little difference to the write miss rates, however.

The lack of any significant variance in the write miss rate when varying the retirement policy for trace1.out could also be explained by the distribution of the write instructions. The writes are seemingly equidistant, meaning that consistent and spaced additions to the write buffer should result in regular writes from the front of the buffer, as soon as the retire-at-N value is reached. There is likely enough latency caused by the reads between the write instructions for each write to complete without causing the next to wait. As the nature of the trace file is very consistent and predictable, varying the value of the retire policy has very little effect.

Additionally, throughout all experiments, the global statistics relating to the ratio of private to shared memory lines remain constant for each trace file, which is to be expected.

To summarise, it would appear that the value of the retire policy does not have a significant impact on the performance of either program, although it could potentially have impact on different types of program. The write-buffer size has a much greater impact, and by increasing the length, the write miss rates are decreased. It appears to be the case that read bypassing is largely responsible for the improvements in all miss rates.