Coursework 2
Cache Coherence

Parallel Architectures

Issued:	Friday February 5, 2021
Due:	Friday March 12, 2021 at 4pm (softcopy on Learn)

1. Introduction

This assignment is the second coursework of the Parallel Architectures course. This coursework will contribute 35% of the final mark for this course.

It consists of a programming exercise culminating in a brief written report. Assessment of this practical will be based on the correctness, clarity and creativity (for the last part) of the solution. This coursework is to be solved individually to assess your competence on the subject. Please bear in mind the School of Informatics guidelines on plagiarism. Please refer to Section 3 for submission instructions.

2. Overview of the Coursework

The goal of this coursework is to develop a cache coherence protocol simulator and evaluate it on the two memory access traces provided. The trace files will be given to you and their format is described in Appendix A.2. To accomplish this goal, you will need to undertake two tasks (Sections 2.1 and 2.2) and document them sufficiently in your report (the report format is discussed in Section 3.2).

2.1. Task-1: Directory-based MSI protocol - 80 marks

The first task is broken down into three subtasks:

2.1.1. Implementation *(30 marks)*

In order to simulate coherent caches, first you will have to write a simple cache simulator. On top of this cache simulator you will build an MSI directory-based coherence protocol with data forwarding on the configuration shown in Figure 1. You will assume only 4 processors with a single level of cache per processor. Given a trace, the simulator will compute the latency of every memory access. Latency will vary based on which component of the memory hierarchy serves the data.

Implementation details. The protocol is a directory-based MSI variant with data forwarding. You must specify and implement your simulator according to the guidelines

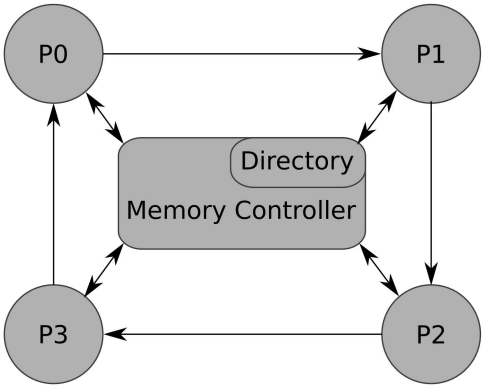

Figure 1: Ring network of processors

of Appendices A and B. Appendix A describes the implementation details of the baseline architecture, the traces and the protocol. Appendix B contains concrete examples illustrating how the latency of a request is computed in various cases.

How to code the simulator. Section 3.1 describes in detail the requirements your code must fulfill.

Statistics. You will have to augment your simulator with mechanisms to collect and report important statistics. The statistics must be reported for the two traces in your written report through a table. In addition, your code will produce an output file (per trace) that contains these statistics. Appendix C enumerates the necessary statistics and describes the exact format of reporting them.

2.1.2. Validation of the simulator *(20 marks)*

In order to have high confidence in the validity of your results, you should provide a small, artificial trace (you must create this trace), which uses the various output options (described in Appendix A.2) to demonstrate that your simulator is working correctly. The artificial trace and the corresponding command line output of your simulator must be included in the written report. The trace you choose is entirely up to you, but you should make sure it covers important cases, such as reads and writes to a cache-line that is in state M or S in another cache. The artificial trace must be included in your submission (inside the archive, see Section 3).

2.1.3. Analysis questions *(30 marks)*.

Inferring information from the statistics you gathered for the two traces, you must answer the following questions (in the report):

1. Characterize the two traces (trace1.txt and trace2.txt), focusing on the factors that increase the average latency. Characterizing the behaviour of a trace entails describing the properties of the access patterns. For example: What kind of misses are occurring? Is there a lot of synchronization? Is there locality and if so what kind of locality?

2. Given the above characterization:

   2.1 How would you optimize the hardware (see Note 2 below) to reduce the average latency for trace1?

2.2 How would you optimize the hardware to reduce the average latency for trace2?

2.3 How would you optimize the hardware to reduce the average latency for both of the traces considered together?

Note 1: additional statistics. When characterizing the traces and proposing optimizations, you can include additional statistics and measurements from your simulator (beyond the mandatory statistics), provided that useful conclusions can be drawn to support your claims and hypotheses.

Note 2: hardware optimization clarification. You are free to add additional hardware structures, make structural changes to the caches or the coherence protocol, or anything else you can imagine. Assume a total budget of 1KB SRAM (per question), noting that you do not have to use it, if it is not needed (Appendix A.4 expands on how the budget can be allocated).

Note 3: optimization motivation. It is crucial that sufficient justification is provided, when proposing an optimization. Attempt to provide critical insight on the workloads of trace1 and trace2, which motivates your proposed optimizations.

2.2. Task-2: Optimization - 20 marks

As part of the analysis questions in Task-1 (§2.1.3), you were asked to propose an optimization for both traces (question 2.3) based on your characterization. You are now asked to implement the optimization and evaluate your hypotheses. Therefore, you must implement your proposed optimization(s) (from question 2.3) attempting to reduce the average latency of the two traces. Appendix A.4 contains more details about your implementation and evaluation of the optimization.

This task is broken into two subtasks:

1. Implement your optimization and report all the statistics, identically to the first subtask of task-1 (Section 2.1.1).

2. Validate your optimization by means of providing a small artificial trace which uses the various output options (identically to Section 2.1.2).

Note 1. If, when answering question 2.3, you concluded that there is no optimization that covers both traces, then you can implement your proposed optimization of either question 2.1 (i.e., for trace1), or 2.2 (i.e., for trace2). *Note however, that if you do propose an optimization for question 2.3, then you must implement it.*

Note 2. To receive marks for Task-2, it is not mandatory to be successful on reducing the latency for both traces. Rather, the goal of this task is to perform a correct, clear and complete evaluation of your hypothesis. The validity and the creativity of your hypothesis itself will be assessed as part of Task-1 (i.e., the analysis questions in §2.1.3).

3. Format of your submission

Submit an archive of your simulator (tarred, zipped) *and* a soft copy of your report (in pdf format), using the Learn environment, similarly to the first coursework. (Go to Assesment;

then Assignment Submission; then coursework-2; Browse local files; select tarred, zipped, simulator archive; then select report in pdf format; then hit Submit).

3.1. Code guidelines

- You are free to use the programming language of your choice for this assignment.
- You must ensure that your code is readable: comment as needed and give descriptive names to functions, variables etc.
- Crucially, your archive must contain a bash script named *run-script.sh* that compiles (if applicable) and executes your code. The script must take as an argument the name of the trace to be run. In addition, if you have implemented an optimization, you must provide an argument to the script that enables the optimization. The optimization should *not* be enabled by default.
- Your code *must* work on DICE.
- Your code must create a command-line output in response to trace output commands as explained in Appendix A.2.2. In addition, your code must generate an output file whose name and format are explained in Appendix C.

3.2. Report guidelines

Your report must contain the following sections:

1. Code details: In this section you should: 1) specify which parts of the assignment are implemented, 2) provide an overview of your source code and how to run it, 3) specify how your optimization(s) can be enabled, when running your code.
2. Task-1: This section should be split in three subsections: 1) Show your statistics for the directory-based protocol for the two traces (as described in Appendix C). 2) Validation: as described in Section 2.1.2. 3) Analysis questions: provide answers to the questions of Section 2.1.3.
3. Task-2: This section should be split in three subsections: 1) A description of your optimization(s). 2) Show your statistics (as described in Appendix C) along with any additional statistics you deem necessary. 3) Validation: as described in Section 2.1.2

4. Reporting Problems

Send an email to Vasilis.Gavrielatos@ed.ac.uk for any issues regarding the assignment.

References

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2011.

Appendices

A. Implementation details

Event	Clock Cycles
Cache Probe (Tag and State access)	1
Cache Access (Read or Write)	1
Extra SRAM Access	1
Directory Access	1
One Hop between Processors	3
One Hop between a Processor and Directory	5
Memory Access latency	15

Table 1: Latency in clock cycles for various events in the network

A.1. Baseline Architecture

Our idealised architecture has 4 processors which are connected via a ring network and a very simple memory hierarchy (shown in Figure 1). Every processor is connected to the memory controller. The directory is maintained adjacent to the memory controller. Memory is addressed at the word level and data addresses start from 0. There is no virtual memory. Thus, address 0 indexes the first word in memory, address 1 the second word, and so on.

The cache-line/block size is 4 words and the private cache of each processor contains 512 cache lines. The caches are direct-mapped with a writeback policy. Please refer to Hennessy and Patterson [1] for details on how addresses (word addresses in the case of this practical) are broken into index and tag.

A.2. Trace File Format

Your simulator *must* accept its input from trace files. Every line in the trace describes either a memory operation, or a command for your simulator to produce some output.

A.2.1. Memory operations

A memory access line in the trace consists of the processor identifier (e.g., `P1`), followed by a space, followed by the type of operation (`R` or `W`, for read or write, respectively), followed by another space, followed by the word address, and ending with a newline. Thus for instance, the line:

```
P2 R 12
```

indicates a read by processor P2 to word 12. Memory operations must appear in the network in the order specified in the trace. Crucially, the memory operations are executed sequentially, one-by-one as found in the trace irrespective of the originating processor. For

Format	Example (Processors: 2 | Cache Size: 2 lines)
<Processor> <cache-line/block number> <tag> <state>	P1 0 13 S 1 8 M P2 0 4 S 1 32 M

Table 2: Possible format for the output command p

example, the second memory operation of the trace will start executing in the simulator only after the first memory operation of the trace is completely processed. Therefore you do not need to model transient states in your simulator.

Note that the actual data value being written is not specified. Thus, your simulator does not (and cannot) keep track of the data that is stored in a cache-line.

A.2.2. Output commands

An output command line in the trace consists of one of the following:

- `v`: indicates that full line-by-line explanation should be toggled, i.e. switched on (if it is currently off) or off (if it is currently on). The default is that it is off.
- `p`: indicates that the complete content of the cache should be output in some suitable format. A sample format is shown in table 2, but you are free to use your own format.
- `h`: indicates that the hit-rate achieved so far should be output (only a memory access that is served in its entirety in the local cache is considered a hit).

You are free to choose the actual wording of the explanations and other output. As a suggestion you can use: "A read by processor P2 to word 17 looked for tag 0 in cache-line/block 2, was found in state Invalid (cache miss) in this cache and found in state Modified in the cache of P1."

For example, the meaning of the following trace file is shown in parentheses, to the right of the commands (these comments are not actually present in the trace):

```
v           (switch on line-by-line explanation)
P0 R 17     (a read by processor 0 to word 17)
P1 R 18     (a read by processor 1 to word 18)
P2 W 17     (a write by processor 2 to word 17)
v           (switch off line by line explanation)
P2 R 25     (a read by processor 2 to word 25)
P0 R 35     (a read by processor 0 to word 35)
p           (print out the cache contents)
h           (print out the hit rate)
```

A.3. Protocol and latency

The directory-based MSI variant, that you will implement, works with data forwarding (and is slightly different that what was discussed in class). When P1 misses on a read to

its local cache, then it contacts the directory; the directory then asks the closest sharer (i.e., closest to P1) to forward the cache-line to P1. Similarly, on a write-miss from P1 (cache-line in S or I state), P1 contacts the directory. The directory then sends invalidations to all sharers, and (in parallel) communicates the number of sharers to P1. The sharers send their acknowledgements directly to P1. If P1 was originally in I state the data must also be forwarded by a sharer, if any, or else from memory. When P1 has received acknowledgements from all sharers, then it can perform its write.

The directory is stored alongside the memory controller. You may assume that the directory maintains the state and the sharer information for all memory blocks (but not the blocks themselves). All processors can directly communicate to the directory, i.e. the distance between every processor and the directory is the same (i.e., 5 cycles). However, communication between processors happens *strictly clock-wise* through the ring. For example P1 requires three hops to send a message to P0. But P0 can send a message to P1 in only one hop. Notably a processor-to-processor hop (3 cycles) is smaller than the hop between directory and processors (5 cycles). If there are multiple sharers for a read miss, the directory will identify the sharer that is closest to the processor issuing the read request. Additionally, for a write miss, the data will also be forwarded along with the invalidation acknowledgements directly to the requesting processor, as before through the shortest path.

You may assume that the invalidations issued to all the caches overlap in time. The invalidation acknowledgements issued to the requester by all sharers also overlap in time. However, acknowledgements may take different number of hops to reach the requesting processor. In that case, the longest route will be considered while calculating latency. Note that, the requesting processor will act as a serialization point while sending invalidation messages. So, the requesting processor will count all the acknowledgements for invalidation messages before writing to the cache-line.

For simplicity, assume that replacements from the cache are notified to directory (to keep sharer vector intact) and write back memory at zero cost. Plainly, the latency effect of write-backs is not measured. Also, assume that the directory has an infinite capacity. So, the directory can maintain state for all the lines in all the local caches.

Table 1 lists the time taken in processor clock cycles for every relevant system event In Appendix B, a number of examples cover in detail how latency must be measured. You must use the examples along with the above description to specify and implement your simulator.

A.4. Custom Optimization

You are afforded a fixed SRAM budget of 1K bytes, which you are free to use in whatever way you like in order to reduce the average latency. You may allocate the entire budget to one component or distribute it among various components in whatever proportion you like. For a given read or write request, accessing this extra SRAM component (or part of component in case your distribute it) will incur a fixed latency of 1 clock cycle. If you choose to allocate the budget towards one or more hardware structures that store data words, then you should assume that a data word is 8 bytes, and manage the 1KB SRAM accordingly. (Note that the size of a data word is not needed otherwise.)

Example	Request	States				Latency
		P0	P1	P2	P3	
B.1	P0 Write	M	I	I	I	2 cycles
B.2	P0 Read	S	I	I	I	2 cycles
B.3	P0 Write	I	I	I	I	29 cycles
B.4	P0 Read	I	I	I	I	29 cycles
B.5	P0 Write	S	I	I	I	14 cycles
B.6	P0 Write	I	S	I	I	25 cycles
B.7	P0 Write	I	S	I	S	24 cycles
B.8	P0 Write	I	I	M	I	22 cycles
B.9	P0 Read	I	S	I	S	19 cycles
B.10	P0 Read	I	S	I	I	25 cycles
B.11	P0 Read	I	I	M	I	22 cycles

Table 3: Summary of the examples

B. Examples of measuring latency

Below, we show how latency must be measured for a few examples. Table 3 provides a summary of the examples.

B.1. P0 performs a Write, local state: M, no sharers

- Probing local cache to match tag and check the state : 1 cycle
- Writing data to local cache : 1 cycle
- Total : 2 cycles

B.2. P0 performs a Read, local state: S, no sharers

- Probing local cache to match tag and check the state : 1 cycle
- Reading data from local cache : 1 cycle
- Total : 2 cycles

B.3. P0 performs a Write, local state: I, no sharers

- Probing local cache to match tag and check the state : 1 cycle
- Send message to directory to request data and invalidate other copies : 5 cycles
- Directory access : 1 cycle
- Directory receives data from memory after finding no on-chip sharers : 15 cycles
- Directory sends data to P0 : 5 cycles

- Probing local cache to change the state : 1 cycle
- Writing data to local cache : 1 cycle
- Total : 29 cycles

B.4. P0 performs a Read, local state: I, no sharers

- Probing local cache to match tag and check the state : 1 cycle
- Send message to directory to request data : 5 cycles
- Directory access : 1 cycle
- Directory receives data from memory after finding no on-chip sharers : 15 cycles
- Directory sends data to P0 : 5 cycles
- Probing local cache to change the state : 1 cycle
- Reading data from local cache : 1 cycle
- Total : 29 cycles

B.5. P0 performs a Write, local state: S, no sharers

- Probing local cache to match tag and check the state : 1 cycle
- Send message to directory to invalidate other copies : 5 cycles
- Directory access : 1 cycle
- Directory responds that there are no sharers : 5 cycles
- Probing local cache to change the state : 1 cycle
- Writing data to local cache : 1 cycle
- Total : 14 cycles

B.6. P0 performs a Write, local state: I, Remote cache-line at P1 in S state

- Probing local cache to match tag and check the state : 1 cycle
- Send message to directory to request data and invalidate other copies : 5 cycles
- Directory access : 1 cycle
- Directory sends message to P1 to invalidate and forward the line : 5 cycles
- Probe cache at P1 to match tag and check the state : 1 cycle

- Access cache at P1 to forward line to P0 : 1 cycle
- Directory sends message to P0 to indicate how many acknowledgements to expect : 0 cycles (this is always overlapped)
- P1 sends invalidation acknowledgement and data to P0 : 9 cycles
- Probing local cache to change the state : 1 cycle
- Writing data to local cache : 1 cycle
- Total : 25 cycles

B.7. P0 performs a Write, local state: I, Remote cache-line at P1, P3 in S state

- Probing local cache to match tag and check the state : 1 cycle
- Send message to directory to request data and invalidate other copies : 5 cycles
- Directory access : 1 cycle
- Directory sends message to P1 to invalidate the line : 5 cycles
- Probe cache at P1 to match tag and check the state : 1 cycle
- Directory sends message to P3 (the closest sharer) to invalidate and forward the line : 0 cycles (latency is overlapped)
- Probe cache at P3 to match tag and check the state : 0 cycles (latency is overlapped)
- Access cache at P3 to forward line to P0 : 0 cycles (latency is overlapped)
- Directory sends message to P0 to indicate how many acknowledgements to expect : 0 cycles (this is always overlapped)
- P1 sends invalidation acknowledgement to P0 : 9 cycles
- P3 sends invalidation acknowledgement and data to P0 : 0 cycles (latency is overlapped)
- Probing local cache to change the state : 1 cycle
- Writing data to local cache : 1 cycle
- Total : 24 cycles

B.8. P0 performs a Write, local state: I, remote cache-line at P2 in M state

- Probing local cache to match tag and check the state : 1 cycle
- Send message to directory to request data and invalidate other copies : 5 cycles
- Directory access : 1 cycle
- Directory sends message to P2 to invalidate and forward the line : 5 cycles
- Directory sends message to P0 to indicate how many acknowledgements to expect : 0 cycles (this is always overlapped)
- Probe cache at P2 to match tag and check the state : 1 cycle
- Access cache at P2 to forward line to P0 : 1 cycle
- P2 sends invalidation acknowledgement and data to P0 : 6 cycles
- Probing local cache to change the state : 1 cycle
- Writing data to local cache : 1 cycle
- Total : 22 cycles

B.9. P0 performs a Read, local state: I, Remote cache-line at P1, P3 in S state

- Probing local cache to match tag and check the state : 1 cycle
- Send message to directory to request data : 5 cycles
- Directory access : 1 cycle
- Directory sends message to P3 (the closest sharer) to forward the line : 5 cycles
- Probe cache at P3 to match tag and check the state : 1 cycle
- Access cache at P3 to forward line to P0 : 1 cycle
- P3 sends data to P0 : 3 cycles
- Probing local cache to change the state : 1 cycle
- Reading data from local cache : 1 cycle
- Total : 19 cycles

B.10. P0 performs a Read, local state: I, Remote cache-line at P1 in S state

- Probing local cache to match tag and check the state : 1 cycle
- Send message to directory to request data : 5 cycles
- Directory access : 1 cycle
- Directory sends message to P1 to forward the line : 5 cycles
- Probe cache at P1 to match tag and check the state : 1 cycle
- Access cache at P1 to forward line to P0 : 1 cycle
- P1 sends data to P0 : 9 cycles
- Probing local cache to change the state : 1 cycle
- Reading data from local cache : 1 cycle
- Total : 25 cycles

B.11. P0 performs a Read, local state: I, remote cache-line at P2 in M state

- Probing local cache to match tag and check the state : 1 cycle
- Send message to directory to request data : 5 cycles
- Directory access : 1 cycle
- Directory sends message to P2 to forward the line : 5 cycles
- Probe cache at P2 to match tag and check the state : 1 cycle
- Access cache at P2 to forward line to P0 : 1 cycle
- P2 sends data to P0 : 6 cycles
- Probing local cache to change the state : 1 cycle
- Reading data from local cache : 1 cycle
- Total : 22 cycles

Statistics	Trace-1	Trace-2
Private-accesses		
Remote-accesses		
Off-chip-accesses		
Total-accesses		
Replacement-writebacks		
Coherence-writebacks		
Invalidations-sent		
Average-latency		
Priv-average-latency		
Rem-average-latency		
Off-chip-average-latency		
Total-latency		

Table 4: Example of Statistics Table

C. Statistics

The key statistics you must report are the following:

1. **Private-accesses**: number of memory access that are served in their entirety in the local, private cache *without* communicating with any other caches/directory.
2. **Remote-accesses**: number of memory accesses that are not private, but can be served on-chip, without contacting memory. Remote-accesses include reads to Invalid state, and writes to Shared and Invalid state, that can be served *on-chip* by means of communicating with the directory (but not main memory).
3. **Off-chip-accesses**: number of memory accesses that are served off-chip from memory.
4. **Total-accesses**: total number of memory accesses (the sum of Private-accesses, Remote-accesses and Off-chip-accesses).
5. **Replacement-writebacks**: number of writebacks generated when a cacheline in M state is written back to memory due to replacement.
6. **Coherence-writebacks**: number of writebacks generated due to the coherence protocol (e.g., when there is a read miss and a cacheline is in M state in a remote cache, the remote cache replies with the data to the originator; this would also generate a writeback of the modified data to keep the main memory up-to-date).
7. **Invalidations-sent**: total number of invalidations sent.
8. **Average-latency** the overall average latency.
9. **Priv-average-latency** the average latency of the Private-accesses.
10. **Rem-average-latency** the average latency of the Remote-accesses.

11. **Off-chip-average-latency** the average latency of the Off-chip-accesses.
12. **Total-latency** The absolute number of the total latency of all accesses.

Note. Statistic are not reported per processor, but rather per trace. Therefore a statistic includes all four processors, for example the Replacement-writebacks statistic refers to *all* Replacement-writebacks by all processors for a given trace.

Presenting statistics in the report. You must create a table similar to Table 4, and fill it with the stats gathered from your implementation for the directory-based protocol (Task-1). Then you must do the same for Task-2.

File output from code. In addition, your simulator must generate an output file that contains all gathered statistics. The output file should be named *out_<trace_name>.txt*. For example, the output file generated when executing trace1.txt should be: *out_trace1.txt*. The output should be as follows (Xs should be replaced with the numbers measured):

```
Private-accesses:  X
Remote-accesses:  X
Off-chip-accesses:  X
Total-accesses:  X
Replacement-writebacks:  X
Coherence-writebacks:  X
Invalidations-sent:  X
Average-latency:  X
Priv-average-latency:  X
Rem-average-latency:  X
Off-chip-average-latency:  X
Total-latency:  X
```