Multicore Architectures Lab Assignments

Tamer Dallou, Jan Lucas, Ben Juurlink

Embedded Systems Architecture

Technische Universität Berlin

Einsteinufer 17, 10587 Berlin, Germany

`{dallou, j.lucas, b.juurlink}@tu-berlin.de`

20. Oktober 2014

This document contains the lab assignments of the Multicore Systems (MCS) course. The five assignments are grouped into two parts. Part 1 contains three tasks related to cache coherence. Part 2 contains two tasks related to GPU programming. It is recommended that you do the assignments in their respective order so that the lab and lecture topics are synchronized.

Each assignments has deliverables which are further described in the assignments. The deliverables have to be submitted via the ISIS page of MCS, which has a link for each assignment in the Grading and Examination section. The assignments can be performed in groups of 3-4 students. Submit only one assignment per group.

Deadlines of the assignments are:

Part 1 Cache Coherence - Task 1	**10.11.2014**
Part 1 Cache Coherence - Task 2	**01.12.2014**
Part 1 Cache Coherence - Task 3	**22.12.2014**
Part 2 GPUs - Task 1	**26.01.2015**
Part 2 GPUs - Task 2	**14.02.2015**

Submitting the deliverables past the deadline reduces your maximum points for the respective assignment. You can submit the assignments until midnight on the day of the deadline.

This part contains assignments 1-3, and is based on the lab assigments of the course *Advances in Computer Architecture*[1] given by Prof. C. R. Jesshope, University of Amsterdam.

Introduction

In the Cache Coherence part of the lab, you will use SystemC to build a simulator of a Level-1 D-cache and various implementations of a cache coherency protocol and evaluate their performance. The simulator will be driven using trace files which will be provided. For information about cache coherencies and memory, please refer to the lecture notes, or see Appendix A in file doc/Appendices.pdf.

The framework for this part of the lab can be downloaded from the ISIS page. Download it to your local home directory, and extract it (tar -xvf part1_student_srcs.tar.gz). This framework contains all documentation, the helper library with supporting functions for managing tracefiles and statistics, the tracefiles, and a Makefile to automatically compile your assignments. Using the Makefile, the contents of each directory under src/ is compiled automatically as a separate target. It already includes directories and the code of the Tutorial, as well as a piece of example code for Task 1.

Trace files are loaded through the functions provided by the helper library (aca2009.h and aca2009.cpp), which is in the provided framework. Detailed documentation for these functions and classes is provided in Appendix C in file doc/Appendices.pdf, but the example code for Task 1 also contains all these functions and should make it self-explanatory. Note that, although later on simulations with up to eight processors with caches are required to be built, it is easier if you start with a single CPU organization. Then, extend it to support multiple CPUs. Make sure that the number of CPUs and caches in your simulation is not statically defined in your code, but uses the number of CPUs read from the trace file. When you are debugging, please print important events on the console or in a log file, in order to evaluate the correctness of each action in your cache system. Part of the evaluation of your work is also based on this log information when you present your work to the lab assistants.

Trace files

There are 3 kinds of trace files available to you: random, debug and FFT. All versions exist for 1, 2, 4 and 8 processors. The random (rnd_pX.trf) trace files have the processors read or write memory randomly in a window in memory that is moving. The debug (dbg_pX.trf) trace files are a short version of the random trace files, suited for debugging your simulation. The FFT (fft_16_pX.trf) trace files are the result of the execution of a FFT on a 64K array and can be used, along with the random trace files, to generate results for your reports. All reads and writes are byte-addressed, word-aligned accesses.

Submission and reports

Students can work in groups of 3-4 persons. Submitting work is through the ISIS page, and includes:

- the source code (packed in tar/gz/zip format) at the end of each task

[1]http://staff.science.uva.nl/ mwvantol/teaching/ACA2009/

- a brief report (in pdf, with cover page indicating group members and assignment name) at the end each task with your results, explanation of results. Moreover, at the end of Task 3, please also include a comparison between the protocols implemented in Tasks 2 and 3.

General guidelines:

- Your source code should compile and run without any modifications on the Linux lab machines, either using the provided Makefile or your own which has to be included.
- The first commandline argument your program accepts is the name of the tracefile. (this happens automatically when the init_tracefile function is used).
- Hit/Miss rate statistics should be gathered with the provided functions, and should be printed with the stats_print function after the simulation ends.
- Your simulations should always terminate and exit without any errors.
- Supporting a different number of CPUs in your simulation should not require recompilation.

Additional Information

Even though attending the lab sessions is not compulsory, we would suggest you to come in regularly. This is because the lab assistants will have the most time for answering your questions during the sessions. If you want to work from home or your own machine and want to be able to test if your code is working properly on the Linux lab machines, you can log in to the student systems from outside using SSH (google tu-berlin sshgate).

If you had any questions for the lab assistant outside the lab time, you can email him, and if necessary visit him at his office after aranging an apointment (by email too).

Assignment start: 20.10.2014 Submission deadline: 10.11.2014

Assignment 1 - SystemC Intro and L1 Cache 25 Points

In this task you must build a single 32kB 8-way set-associative L1 D-cache with a 32-Byte line size. The simulator must be able to be driven with the single processor trace files. Assume a Memory latency of 100 cycles, and single cycle cache latency. Use a least-recently-used, write-back replacement strategy.

Hints:

- Read doc/SystemC_tutorial.pdf file, and implement the examples mentioned there.
- You can use the example code provided for Task 1, which is based on the tutorial, as a guide and modify the Memory module so that instead of RAM, it simulates a set-associative data cache.
- As we only simulate accesses to memory, it is not necessary to simulate any actual contents inside the cache. Furthermore, you do not need to simulate the actual communications with the memory, just the delay.

Assignment start: 10.11.2014 Submission deadline: 01.12.2014

Assignment 2 - Valid-Invalid Protocol 45 Points

Extending the code from Task 1, you are to simulate a multiprocessing system with a shared memory architecture. Your simulation should be able to support multiple processors, all working in parallel. Each of the processors is associated with a local cache, and all cache modules are connected to a single bus. A main memory is also connected to the bus, where all the correct data is supposed to be stored, although, again, the memory and its contents does not have to be simulated. Assume a pipelined memory which can handle a request from each processor in parallel with the same latency.

In order to make cache data coherent within the system, implement a bus snooping protocol. This means that each cache controller connected to the bus receives every memory request on the bus made by other caches, and makes the proper modification on its local cache line state. In this task, you should implement the simplest VALID-INVALID protocol. The cache line state transition diagram for this protocol is shown in Figure 1.

Using your simulator you must perform experiments with different numbers of processors (1 to 8 processors) using the different trace files for each case.

For each experiment you should record (but not limited to) the following data for your report:

1. Cache hit and miss rates (use the framework functions)
2. Main memory access rates
3. Average time for bus acquisition (as a measure for bus contention)
4. Total simulated execution time

You should also investigate what happens to cache hit and miss rates when snooping is deactivated.

Hints:

- Some help on implementing a Bus is provided in Appendix B in file doc/Appendices.pdf. You could use or modify these implementations for your simulation.
- The bus can only serve one request at any certain time. Thus if one cache occupies the bus, the other cache has to wait for the current operation to complete before it can utilize the bus. The cache does not occupy the bus when it waits for the memory to respond. The responses from memory should have priority on the bus.
- The cache must not only be able to serve the request from memory, it also has to observe all the transactions happening on the shared bus, and make proper adjustment on its own corresponding cache line states as described in Figure 1.
- Keep in mind you also need to implement cache to cache transfers for when a CPU reads from or writes to a location for which another cache holds data which was not written back to memory yet

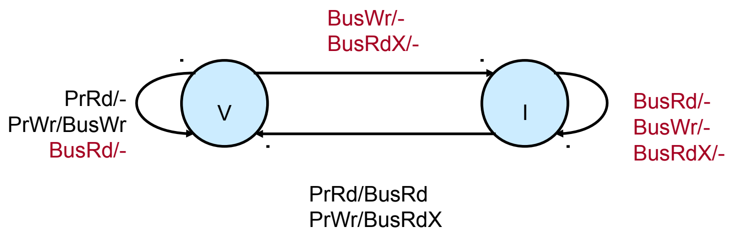


Abbildung 1: VALID – INVALID Cache Line State Transition Diagram

Assignment start: 01.12.2014 Submission deadline: 22.12.2014

Assignment 3 - MOESI Protocol 30 Points

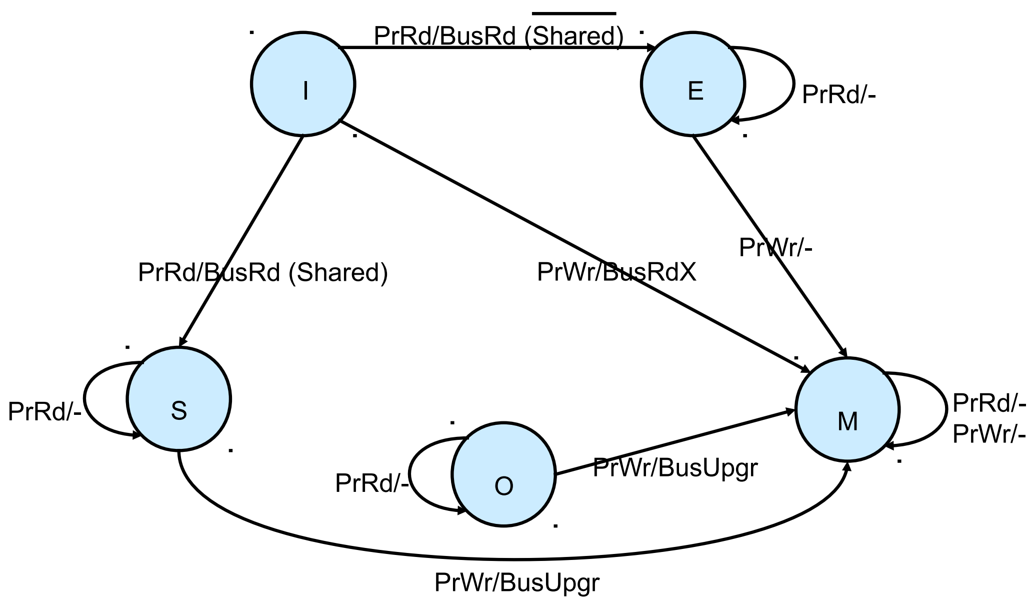
In this task, you are to simulate a version of the MOESI protocol. This protocol is implemented in AMD64 caches. The cache line states of the MOESI protocol are:

- *Invalid*: A cache line in the invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or another processor cache.
- *Exclusive*: A cache line in the exclusive state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.
- *Shared*: A cache line in the shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. If no other processor holds it in the owned state, then the copy in main memory is also the most recent.
- *Modified*: A cache line in the modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.
- *Owned*: A cache line in the owned state holds the most recent, correct copy of the data. The owned state is similar to the shared state in that other processors can hold a copy of the most recent, correct data. Unlike the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state—all other processors must hold the data in the shared state.

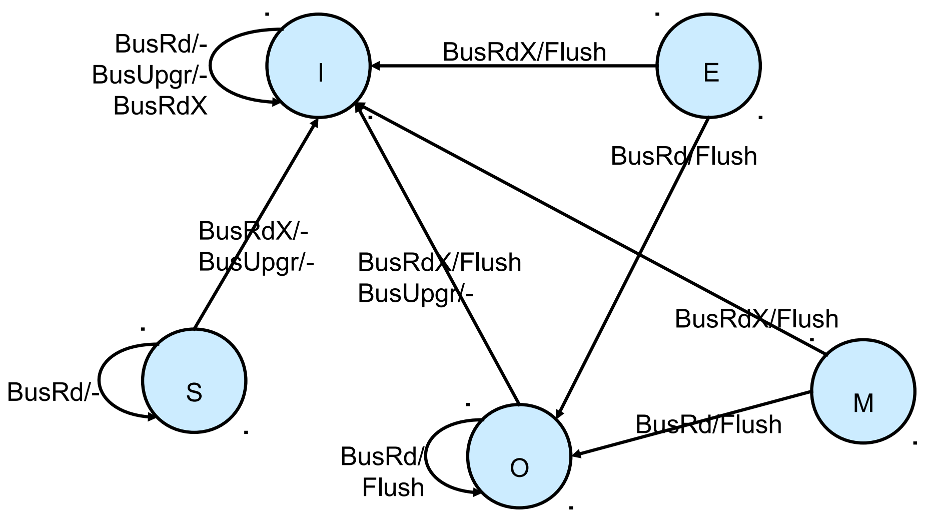
The state transition diagram is depicted in Figure 2

Hints:

- The bus implementation might have to be modified to provide additional information. For instance, when a cache is suffering a read miss, and fetching the data from main memory. It has to check whether the data is incorrect in the main memory (when a cache has a piece of data stored at Modified or Owned state, the value is incorrect in the main memory).
- In case the cache is running out of space, any data in the Modified and Owned state has to be saved to main memory first, before it can be replaced.



MOESI State Transition Diagram for Processor Requests



MOESI State Transition Diagram for Bus Requests

Abbildung 2: MOESI Cache Line State Transition Diagram