CS4223: Multi-Core Architectures

Mini-Project (25 marks)

Deadline – Part 1: Friday 25th of October 2024, 11:59pm
Deadline – Part 2: Friday 15th of November 2024, 11:59pm

The goal of this mini-project is to strengthen your understanding of cache coherence protocols and how simulators are developed.

You are welcome to do this project in groups of at most 2 students. Please let me know if you have difficulty finding a partner.

Please feel free to discuss your doubts in the discussion forum.

Benchmark Traces:

You need to implement a simulator for cache coherence protocols and then explore different design choices using Python, C, C++, C#, Java, Javascript or any other programming language of your choice. Therefore, you can program on any platform of your choice.

Unlike Simplescalar simulator, which was a functional simulator that could execute the benchmarks directly, in this assignment you will develop trace-driven simulator.

You will use three benchmark traces from the PARSEC benchmark suite for this assignment.

- **blackscholes:** Option pricing with Black-Scholes Partial Differential Equation (PDE)
- **bodytrack:** Body tracking of a person
- **fluidanimate:** Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method

The benchmark traces are available from the SoC Compute Cluster:
/home/course/cs4223/assignments/assignment2

For space reasons, the traces have been compressed: Using WinZip (or gunzip), decompress the files in some directory, and you will obtain the trace files, "*.data".

The trace file "benchmarkName_four.gz" contains the trace for multi-core with 4 cores. For example, *blackscholes_four.gz* contains the trace for multi-core with 4 cores for blackscholes benchmark. After running WinZip on *blackscholes_four.gz*, you will get the 4 trace files, "*blackscholes_N.data*", --- one corresponding to each core: *blackscholes_0.data*, *blackscholes_1.data*, *blackscholes_2.data*, and *blackscholes_3.data*. The same applies for *bodytrack_four.gz* and *fluidanimate_four.gz* files.

The trace files, which have the extension ".data", consists of lines where each one has two numbers, separated by only one white space:

Label Value

• **Label** is a decimal number that identifies the operation type executed by the core: *load (read) instruction* (**0**), *store (write) instruction* (**1**) or *other instructions* (**2**). *Other instructions* mean instructions except for memory access (*load/store*) instructions, such as computation instructions (addition, multiplication, division). *Other instructions* only appear between two memory access operations.
• **Value** is a hexadecimal number. For *load* (**0**) and *store* (**1**) instructions, *value* indicates the effective address of the memory word to be accessed by the core, while for *other instructions* (**2**), *value* denotes clock cycles required by other instructions between two memory access operations (*load/store* instructions). **For example, "2 0xc" indicates that the required number of clock cycles by *other instructions* between load/store instructions is 12 (0xc).**

```
0 0x817b08
2 0xc
0 0x817b10
2 0x19
1 0x817b58
2 0x27
1 0x817b00
```

The figure above illustrates an example of trace file generated by one core. The part of the file in the figure shows a trace with two data reads and two data writes. Notice that the trace does not contain actual data as it is not necessary for simulating cache structures.

Assumptions:

You have to make the following assumptions.

1. Memory address is 32-bit. Note that the address shown in the example trace file is 24 bits because the 8 most significant bits are sometimes 0. So the address 0x817b08 is actually 0x00817b08
2. Each **memory reference accesses 32-bit (4-bytes) of data.** That is word size is 4-bytes.
3. We are only interested in the data cache and will not model the instruction cache.
4. Each processor has its own L1 data cache.
5. L1 data cache uses **write-back, write-allocate** policy and **LRU replacement** policy.
6. L1 data caches are kept coherent using cache coherence protocol.
7. Initially all the caches are empty.
8. The **bus uses first come first serve arbitration policy** when multiple processor attempt bus transactions simultaneously. Ties are broken arbitrarily.
9. The L1 data caches are backed up by main memory --- there is no L2 data cache.

10. L1 cache hit is 1 cycle. Fetching a block from memory to cache takes additional 100 cycles. Sending a word from one cache to another (e.g., BusUpdate) takes only 2 cycles. However, sending a cache block with N words (each word is 4 bytes) to another cache takes 2N cycle. Assume that evicting a dirty cache block to memory when it gets replaced is 100 cycles.
11. You may need to make additional assumptions. Clearly state those assumptions in your report.

Also assume that the caches are blocking. That is, if there is a cache miss, the cache cannot process further requests from the core and the core is completely halted (does not process any instructions). However, the snooping transactions from the bus still need to be processed in the cache.

In each cycle, each core can execute at most one memory reference. As per our assumptions, you do not need to model L1 instruction cache. So the instruction address trace is not included. But the core cycle counter still has to be incremented with the cycle value for other instructions in between two load-store instructions. For example, in the example trace file, after data address 0x00817b08 has been processed, the cycle count for the core should be incremented by 12 before the next data address can be processed. This ensures realistic simulation of multi-cores where data accesses from different cores are punctuated with instruction processing.

Executable Program Details

Your program should take the input file name and cache configurations as arguments. The command line should be
coherence *"protocol"* "*input_file*" "*cache_size*" *"associativity"* "*block_size*"
where coherence is the executable file name and input parameters are

- "protocol" is either MESI or Dragon
- "*input_file*" is the input benchmark name (e.g., bodytrack)
- "*cache_size*": cache size in bytes
- *"associativity":* associativity of the cache
- "*block_size*": block size in bytes

For example, to read bodytrack trace files and execute MESI cache coherence protocol with each core containing 1K direct-mapped cache and 16 byte block size, the command will be

coherence MESI bodytrack 1024 1 16

Assume default parameters as 32-bit word size, 32-byte block size, and 4KB 2-way set associative cache per processor.

Your program should generate the following output:

1. Overall Execution Cycle (different core will complete at different cycles; report the maximum value across all cores) for the entire trace as well as execution cycle per core
2. Number of compute cycles per core. These are the total number of cycles spent processing other instructions between load/store instructions
3. Number of load/store instructions per core
4. Number of idle cycles (these are cycles where the core is waiting for the request to the cache to be completed) per core
5. Data cache hit and miss counts for each core
6. Amount of Data traffic in bytes on the bus (this is due to bus read, bus read exclusive, bus writeback, and bus update transactions). Only include the traffic for data and not for address. Thus invalidation requests do not contribute to the data traffic.
7. Number of invalidations or updates on the bus
8. Distribution of accesses to private data versus shared data (for example, access to modified state is private, while access to shared state is shared data)

Basic Tasks – Part 1 (due October 25th):

Implement a single-core CPU and cache hierarchy (cache and DRAM). Be sure to properly implement the bus, timing, bandwidth, and DRAM latencies. Use only the first dataset when reading and writing values to the cache (for example, blackscholes_0.data).

Submit (1) code (and Makefile (or equivalent) to allow building), and (2) a short report with output examples of your code running.

Basic Tasks – Part 2 (due November 15th):

Implement MESI invalidation-based cache coherence protocol and Dragon 4-state update-based cache coherence protocols as discussed in class and run it on the trace files provided.

Quantitative Analysis:

1. Compare MESI and Dragon protocol along multiple dimensions and explain which protocol is better for each benchmark.

Advanced Task:

You may perform a quick literature survey to identify some optimizations to the basic MESI or Dragon protocol. Implement one optimization and evaluate the improvement if any.

Report:

Write a report on your finding and explain your findings.
In your report, you should have a clear introduction and conclusion, describe the programming language and environment you used. Your report should have explanation of your implementation (e.g., data structure, flow chart, etc.). You should then clearly present and analyze the experimental result.

Upload your code and report to Canvas -> Assignments -> Assignment 2.

Your group will be scheduled for a demo of your program and code review during the reading week (right after the submission deadline listed at the top of this document).

Grading:

- Correct implementation of the coherence protocols (includes optional demo): 14 marks
    - MESI Coherence protocol: 7 marks
    - Dragon coherence protocol: 7 marks
- Quantitative Analysis: 4 marks
- Advanced Task: 4 marks
- Report: 3 marks