CS4223 Multi-Core Architectures

Quad-core Cache Simulator with MESI, Dragon, and MOESI Coherence Protocols

Sharadh Rajaraman (A0189906L)
Mohamed Yousuf Minhaj Zia (A0XXXXXXB)

Friday 19th November 2021

Contents

1 Introduction

In a bid to accelerate and exploit certain properties of their workloads, modern computers have evolved several different levels of parallelism—instruction, thread, data and task, loop, etc. One implementation of task-level parallelism is multi-core processing, where *one* processor chip contains several different execution units, or *cores*. These multi-core processors are also called chip multiprocessors (CMPs).

The memory hierarchy model used in classical single-core systems may be extended straightforwardly to CMPs, with the processor itself having several *levels* of cache, each larger and slower than the previous. As always, the *nearest* cache level separates the instruction and data cache. For instance, **Figure 1** depicts the cache layout for a recent notebook Intel Xeon processor.

Given the separation of the cores, the problem of *coherence* is encountered: how are accesses (either reads or writes) by one core seen by the others? These require *coherence protocols*, which dictate the movement and broadcast of accesses from one core to all others. There are two broad categories of coherence protocols: *snooping*, or *bus-based*, and *directory-based*. The focus of this experiment will be simulating three snooping protocols: MESI, Dragon, and MOESI, and analysing their performance.

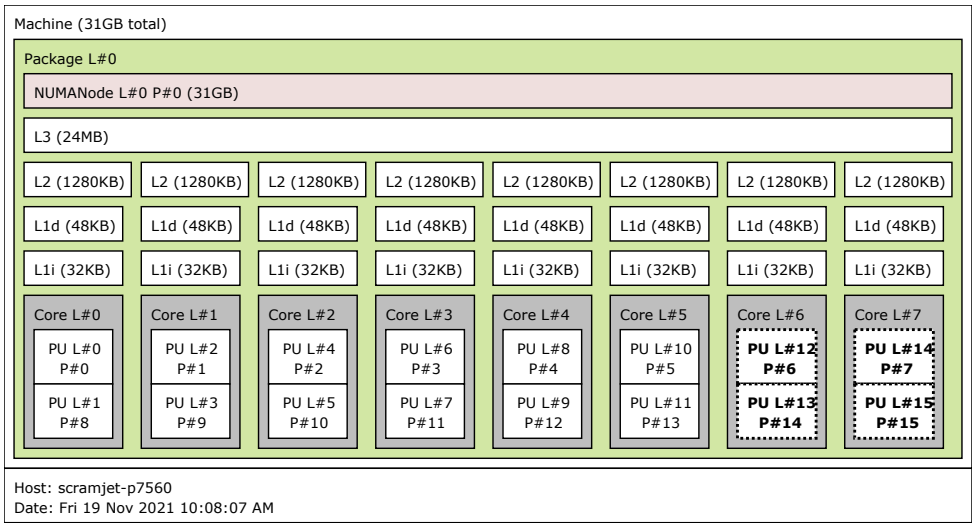

Figure 1: `lstopo -no-io` output for 8-core, 16-thread Intel® Xeon® W-11955M processor

1.1 Snooping Coherence Protocols

In snooping protocols, memory accesses by every core are *broadcasted* into a bus. The cache controllers of all other cores *snoop* on or *listen* to this bus, and change the states of their caches appropriately. Furthermore, in update-based protocols like Dragon, the bus also provides inter-cache lines, along which updated data may be transmitted. Each protocol tested in this assignment has different characteristics and optimises for different use-cases (writeback-heavy, inter-cache transfer-heavy).

1.1.1 MESI Protocol

MESI is an invalidation-based protocol, containing four states: **M**odified, **E**xclusive, **S**hared, and **I**nvalid. There are no cache-cache transfers. As such, all misses for a block are facilitated by accessing the main memory. A cache accessing a block in the **M** state in some other cache would have to wait for the latter cache to write back that block to memory, and incur an additional wait penalty.

	M	**E**	**S**	**I**
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

Table 1: Permitted cache states

Figure 2 shows the MESI state transition diagram, and **Table 1** shows valid states between a given pair of cache lines mapping to the same memory block.

1.1.2 Dragon Protocol

Dragon is an update-based protocol that drastically improves on MESI, by allowing inter-cache transfers. There are four states: **E**xclusive, **S**hared **c**lean, **S**hared **m**odified, and **M**odified. **Figure 3** depicts the state transitions of the Dragon protocol. Note that we have additionally implemented the **I**nvalid state, to represent a clean cache with *no* data whatsoever.

Furthermore, **Table 2** shows valid states between a given pair of caches mapping to the same memory block, under the Dragon protocol.

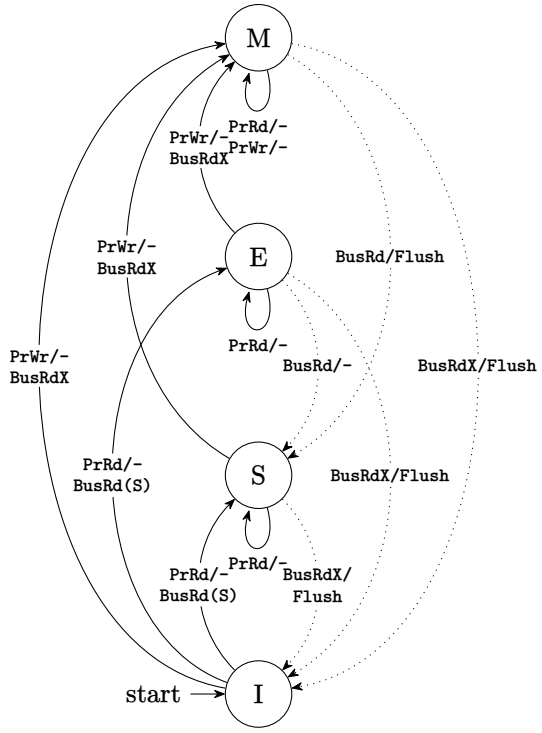

Figure 2: MESI protocol states and transitions

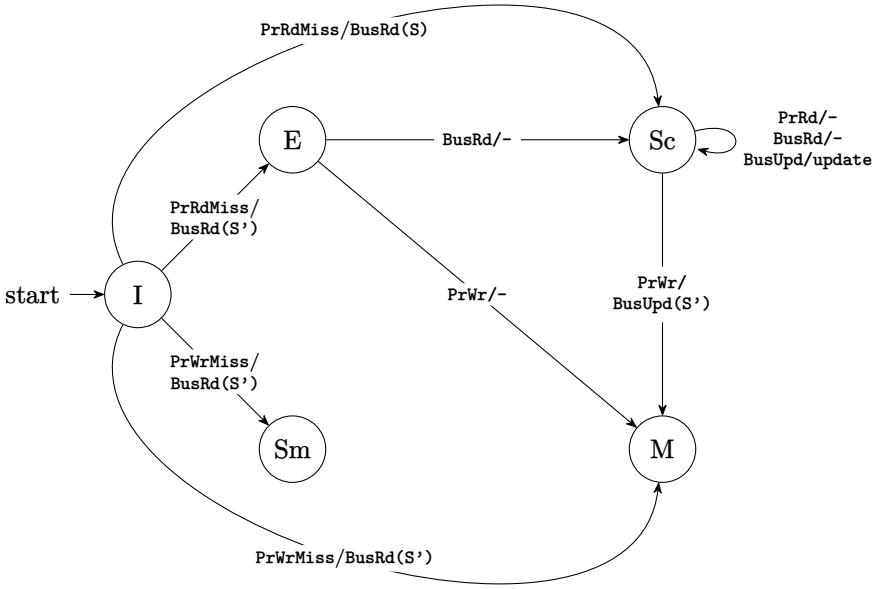

Figure 3: Dragon protocol states and transitions

	E	Sc	Sm	M
E	✗	✗	✗	✗
Sc	✗	✓	✓	✗
Sm	✗	✓	✗	✗
M	✗	✗	✗	✗

Table 2: Permitted cache states for MESI

1.1.3 MOESI Protocol

2 Experimental Design

2.1 Programming Language and Build Tools

Based on our own programming experiences, we had the choices of C++ and C#. We chose C++20 because it allowed us access to several functional libraries like `<ranges>`. We also selected it for its speed, given that truly large cache sizes might have needed to be simulated. Indeed, several configurations of the benchmark took nearly two hours to complete, which might have been far slower had we used C#.

We also used CMake as a build-generator, so that cross-platform programming could be easily done. CLion and `nvim` were used across Windows and Linux to develop the simulator.

2.2 Design Abstractions and Implementation

We have chosen an object-oriented design for the simulator, with each component focused on one task.

2.2.1 Overall System Design

The `Runner` class is responsible for accepting input from the command-line, parsing and validating said input, and forwarding it to the `System` class. `Runner` also collects data from the `CoreMonitor`s and the `BusMonitor`, prints them to `stdout` and writes to a `.csv` file for data analysis. As such, `Runner` also contains `int main()`.

`System` contains an array of four `Processor`s, as well as a 'smart pointer' (specifically, `std::unique_ptr`) to a `Bus` (or more specifically, one of its derived classes). Each `Processor` contains another smart pointer to a `Cache`, whose ownership is shared by the `Bus`, so that the latter may update the caches upon any transition being triggered.

2.2.2 Input Parsing and Validation

We first check that the number of arguments to the program is correct;

2.2.3 The `System` Class

`System::run()` is the main entry point for the

2.2.4 The `Processor` Class

2.2.5 The `Bus` Class and Derived Classes

the different protocols implemented as `MESIBus`, `DragonBus`, and `MOESIBus`, which all inherit from a base class, `Bus`. This allowed the rest of the components to be agnostic to the currently-running protocol. We were also able to easily extract out common functionality for the caches, and

3 Results and Analysis

For all the following graphs, we will use the legend above.

3.1 Cache Size

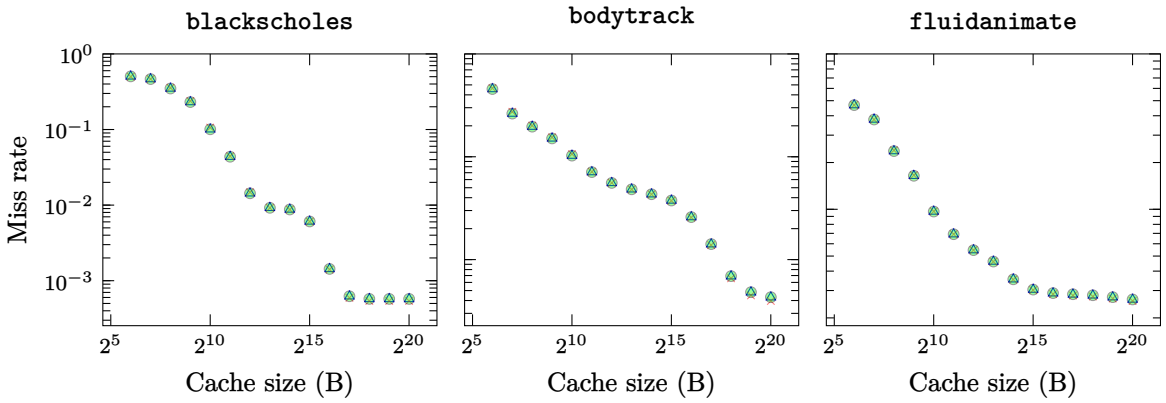

Figure 4: Miss rate on cache size, with fixed block size (32 B) and 2-way set-associativity

3.2 Associativity

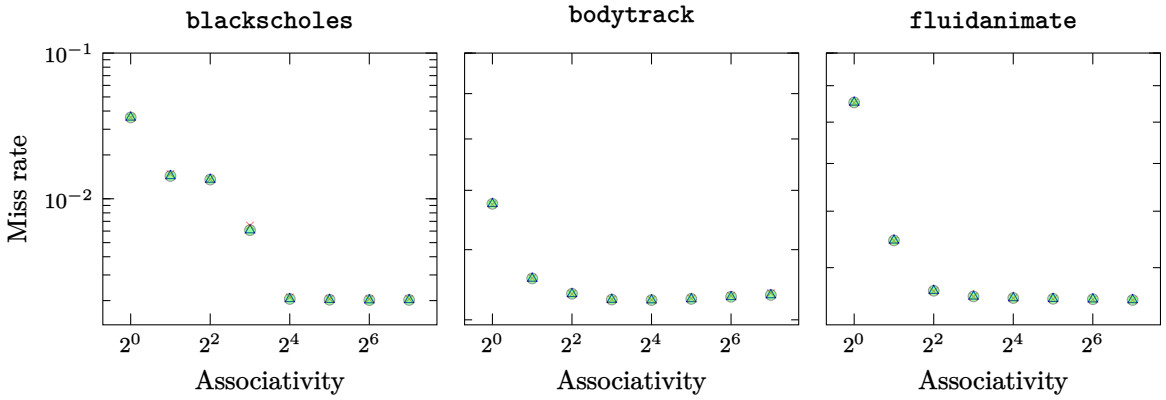

Figure 5: Miss rate on associativity with fixed cache size (4 KiB) and fixed block size (32 B)

3.3 Block Size

4 Conclusion

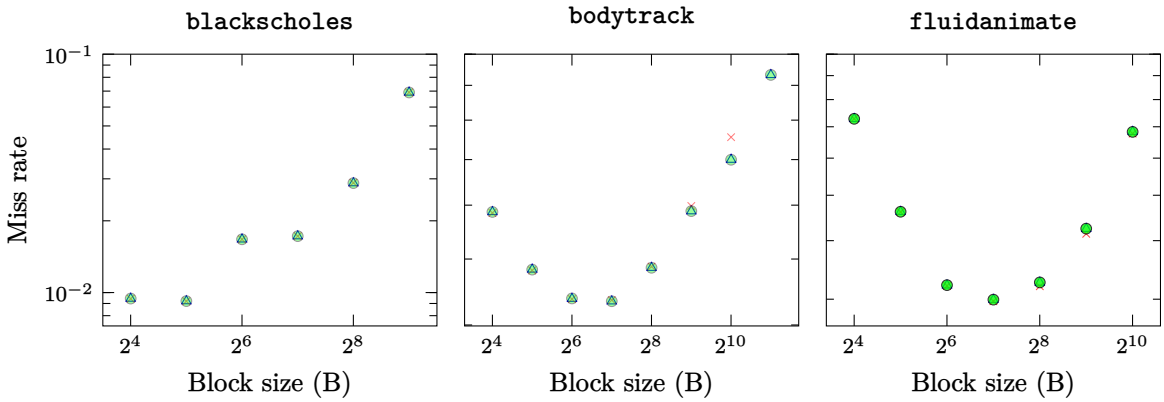

Figure 6: Miss rate on cache line size with fixed cache size (128 KiB) and 2-way set-associativity