CS4223 Multi-Core Architectures

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

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

19 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

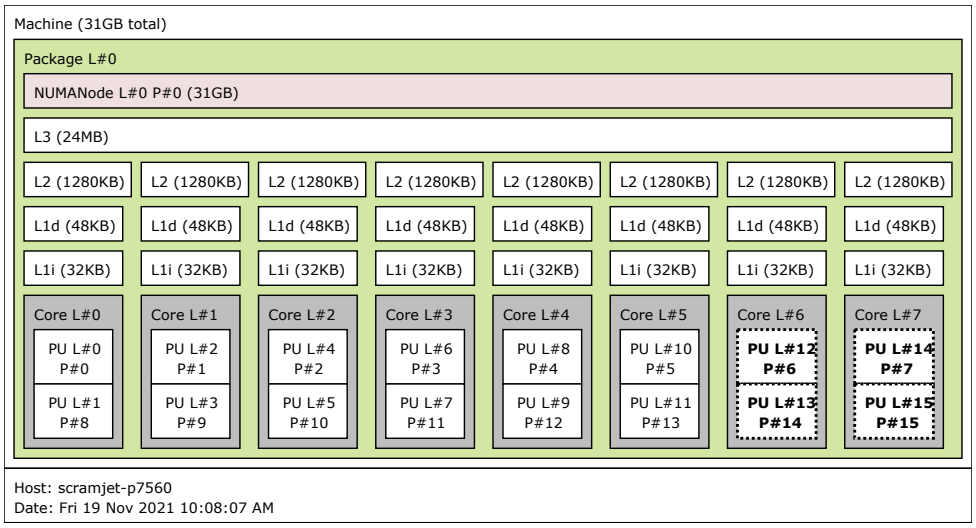

Figure 1: `lstopo -no-io` output for 8-core, 16-thread Intel® Xeon® W-11955M processor

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.

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.

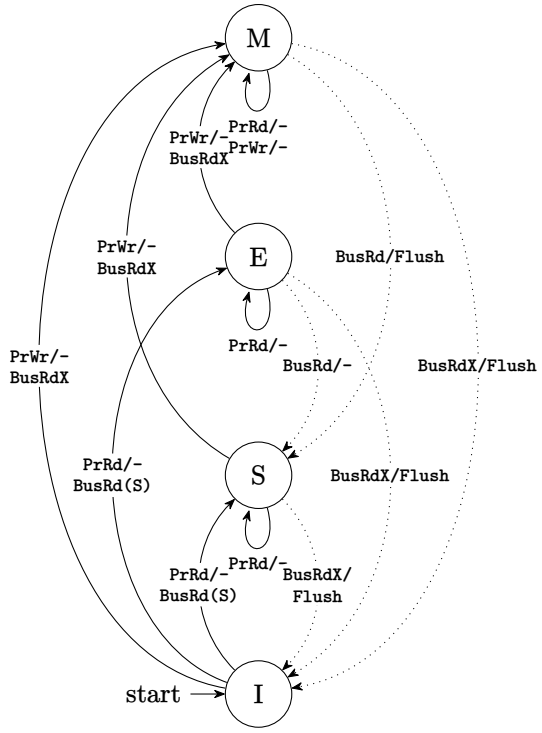

Figure 2: MESI protocol states and transitions

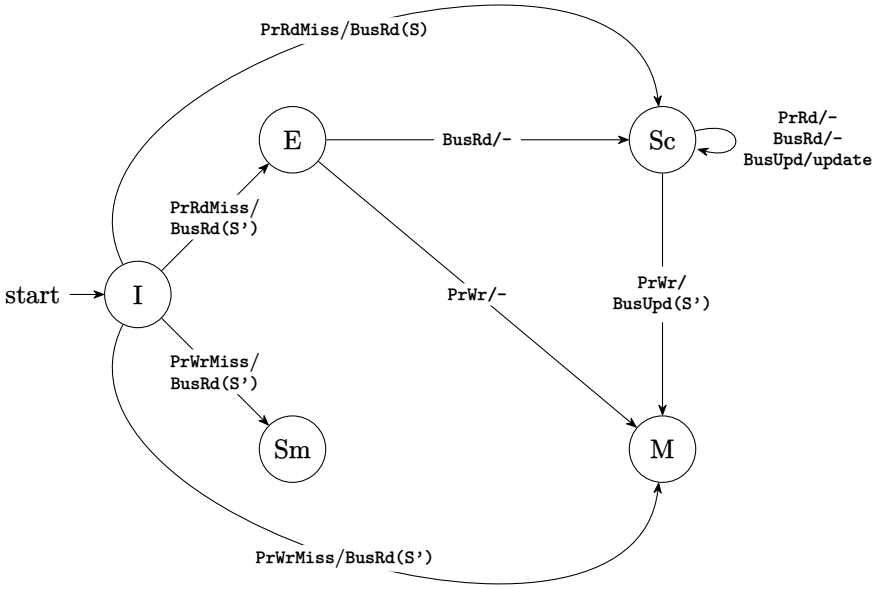

Figure 3: Dragon protocol states and transitions

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

Table 2: Permitted cache states for MESI

1.1.3 MOESI Protocol

The MOESI protocol is similar to the MESI protocol, in that writes cause invalidation to all the other lines in the other caches. However, it introduces another state, **O**wned, that the cache can go into when it encounters a BusRd making it the responsibility of the owning Cache to sharing the line without writing back to memory. Hence, unlike the MESI protocol, dirty inter-cache sharing is allowed, which signicantly reduces read latency from 100 (main-memory reads) to $2N$ where N is the number of words, when an owning cache exists. This drastically reduces the number of main-memory reads in programs that exploit much inter-process communication.

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 accurately, 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. At most 5 arguments are expected, and the defaults are used if fewer than 5 arguments are provided. An exception is thrown if the number of arguments is too high. Next, we check if the protocols and benchmark names are correct; otherwise, an exception is thrown. Valid protocol strings are `"MESI"`, `"Dragon"`, and `"MOESI"` (the checks are case-*sensitive*).

Finally, the cache configuration is checked. The cache size and block size must be some multiple of 4 (the word size), and the cache itself must be a multiple of the block size. The associativity must then evenly divide the number of blocks, ie the relationship in **equation (1)** must be satisfied.

$$\text{associativity} \equiv 0 \pmod{\text{number of blocks}} \tag{1}$$

If any of the above conditions fail, the simulator refuses to run, throwing an exception.

2.2.3 The `System` Class

`System::run()` is the main entry point for the program, and launches a loop that runs indefinitely, as long as any one processor signals that there are still instructions left to process. This is achieved by setting a member variable appropriately. In every iteration of the loop, one processor is refreshed, and any blocked cycle counters are decremented.

If a processor is already *not* blocked to begin with, then the `Bus` triggers state transitions, and the next instruction is streamed in and checked for its type (`LD`/`ST`/`ALU`). Memory access instructions are then checked against the cache for hits or misses, and the bus transitions all other caches appropriately.

2.2.4 The `Processor` Class

`Processor` is an abstraction for a system processor. That said, in our implementation, it does little more than provide a convenient container for various counters, the instruction input stream, and the cache itself. The processor

2.2.5 The `Bus` Class and Derived Classes

`Bus` is an abstract class which calculates the number of cycles the processors need to be blocked for, and also facilitates state transitions for each cache, at the end of the wait period. The three protocols are 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. The derived classes perform the state transitions based on their respective protocols and also find out the block periods of a cache based on the state of all the caches in case of access misses, based on the protocol.

`MESIBus` returns a 100 cycle block on any miss, and 1 cycle latency for hits, as no cache to cache sharing is allowed. `DragonBus` returns a 2 cycle block for write hits due to word updates, but incurs a $2N$ cache latency for inter-cache block sharing on misses, and a 100 cycle latency if no other cache contains the required information. `MOESIBus` forms a middle ground by allowing dirty inter-cache sharing with $2N$ cache latency, but returns a 100 cycle block when no other cache contains the block in question.

2.2.6 The `Cache` Class

`Cache` is our implementation of an least-recently used (LRU) write-back, write-allocate cache. On each write and read operation, the set containing the block is sorted in the order of the latest-accessed line. On write misses, the missing block is loaded into the respective set. Given the *data* itself is not simulated, no modifications are performed, but the state is changed appropriately.

Evictions are picked up by the cache on misses and ejected from the respective set. Control is then returned to `Bus`, which then performs a state transition based on the protocol. Writes-back of dirty lines invoke a 100 cycle block.

2.3 Assumptions Made

There were a few assumptions we have made in our design of the simulator:

1. Instruction fetch and decode takes zero cycles.
2. There is no pipelining in this simulator: the next instruction is fetched and parsed only when the current instruction has completed running.
3. Bus transactions are issued *after* data is received, in case of an access miss.
4. Intermediate bus transactions cause short-circuiting. In other words, suppose a cache A requests block x at time t_1. Cache B requests the same block at time $t_2 > t_1$. Our implementation is such that the latter cache would only have to wait for, at a minimum, $t_2 - t_1$ time before it could access the block. The inter-cache delay is also accounted for, in the case of the Dragon and MOESI protocols.

3 Results and Analysis

△ MESI × Dragon ● MOESI

For all the following graphs, we will use the legend above. We have chosen to plot the miss rate against three independent variables: the cache size, associativity, and the block size. A Python script was used to run the compiled binary with the appropriate arguments, and more than 7000 inputs were generated.

We have chosen some particular results that clearly depict changes in performance, and the metric we have chosen is the miss rate. We chose not to use execution time as it is itself dependent on the miss rate (or more particularly, the miss *count*).

3.1 Cache Size

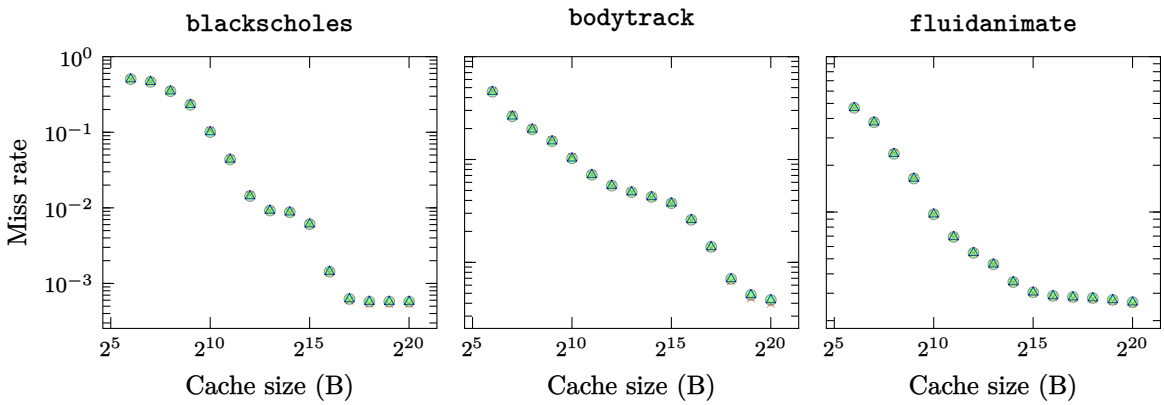

Figure 4: Miss rate on cache size, with fixed block size (32 B) and 2-way set-associativity

From **Figure 4**, it is obvious that all caches show the same behaviour: increasing the cache size results in a corresponding improvement in performance, as the miss rate decreases from 40 % to less than 0.5 % or so. This improvement is consistent across all three cache protocols.

3.2 Associativity

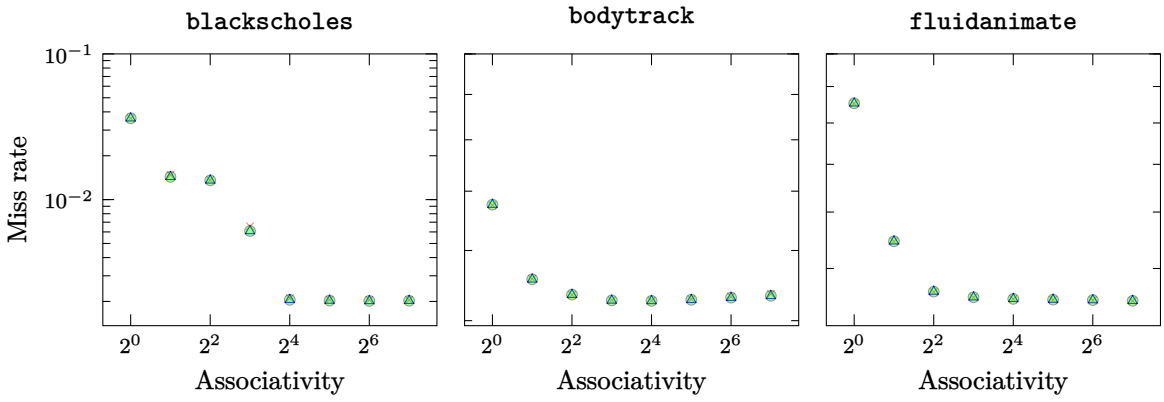

Figure 5: Miss rate on associativity with fixed cache size (4 KiB) and fixed block size (32 B)

From **Figure 5**, it is clear that increasing the associativity of a cache beyond a certain limit provides no real benefit, as diminishing returns are reached. In the real world, most processor caches do not have associativity counts beyond 8 or 16, as the real limit of compulsory misses is hit. This means that CPU manufacturers can leave some performance on the table if it means drastically simplified hardware costs.

3.3 Block Size

Varying block sizes provide the most interesting results so far, as seen in **Figure 6**. Across all caches, miss rates decrease as block sizes increase. There is then a turning point at around 2^7B, beyond which miss rates drastically increase with increasing block sizes. This is because very large block sizes incur big

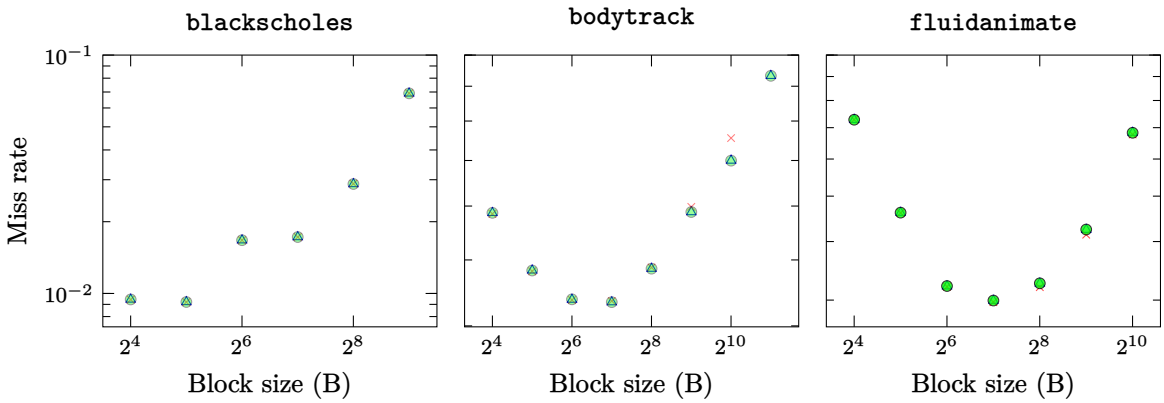

Figure 6: Miss rate on cache line size with fixed cache size (4 KiB) and 2-way set-associativity

penalties in inter-cache transfers, as the large block needs to be loaded completely on the inter-cache line, and sent to the other caches. The `bodytrack` and `fluidanimate` benchmarks show this behaviour most clearly.

3.4 Protocols

From our tests, it is clear that the Dragon protocol outperforms MESI and MOESI in all benchmarks (as expected). Having zero invalidations with inter-cache sharing (notwithstanding an inconsequential 2 cycle block) allows more lines to be retained, hence reducing the miss rate and significantly boosting performance. However, this comes at a price: there is heavy bus traffic due to the BusUpdate word shared. In the real world, this might equate to large amounts of heat dissipated on-chip, even if performance is only *acceptable*.

MESI is the worst protocol because it requires writes back for every BusRd. As already mentioned, there is a default 100-cycle block on writes back, which leads to its poor performance.

Finally, MOESI is an improvement on MESI as dirty inter-cache sharing is allowed. This reduces processor blocks on misses from 100 to 2N in cases when another caches contain a dirty line. Notably, this metric means that there is a point at which $2N \gg 100$, when it then makes more sense for the caches to write back and then retrieve data from the main memory. This doesn't reduce the number of invalidations but in large programs, may reduce the overall execution cycles. However, it still performs worse than Dragon due to the invalidations. Furthermore, MOESI does not allow for clean line cache to cache sharing.

All that said, if bus traffic were taken into account, this protocol would provide a competitive execution cycle to traffic data ratio compared to Dragon and MESI.@

4 Conclusion

This experiment allowed us to understand and simulate cache coherence, as well as compare their performances.

We confirm the straightforward idea that larger caches lead to increased performance, but associativity only improves performance to a limited extent. More importantly, we were surprised by the results for block sizes, that led us to further research and justifications for said results.