CS4223 Mini-Project

Cache Coherence Simulator

Sherman Lim Jun Hong (A0204752L)
National University of Singapore
18 November 2022

Contents

1 Introduction

1.1 Overview

In this project, I designed and implemented a trace-driven Cache Coherence Simulator to explore various design decisions of cache coherence protocols. The Simulator takes input benchmark traces that describes load, store, and compute operations taken by each of the four simulated cores. Then, it runs a simulation assuming a multi-core system with private L1 data caches and a single shared memory. Various measurements, such as cache miss rates and total execution time, are taken during the simulation to allow for comparison between protocols.

Experiments were also conducted to compare the MESI, Dragon, and Illinois MESI protocols using the *blackscholes*, *bodytrack*, and *fluidanimate* benchmarks from the PARSEC benchmark suite. Cache size, associativity, and block size were also varied. The results are presented in this report.

1.2 Programming Language

The programming language of choice for the Simulator is C++20. C++ was chosen for its high performance and ability to create low-overhead abstractions. Choosing a performant language allows the Simulator to handle high load and run realistic benchmarks in a reasonable amount of time. Moreover, the ability to create low-overhead abstractions allows us to create abstractions to help with ensuring correctness and maintainability without compromising performance.

Python was also used for scripting.

1.3 Environment

The operating system used is macOS Monterey 12.6 on an Intel Mac. The compiler used is Apple clang version 14.0.0, which supports the C++20 features used in the Simulator.

2 Implementation

2.1 Overview

An object-oriented approach was used to implement the Cache Coherence Simulator, where classes were created to model after an actual system. An overview of the key classes are described below:

1. **Simulator.** This is the main class that integrates the other components. It processes events (occurrences of core operations or bus transactions) by making use of the other components, which might in turn add more future events. The simulation completes when all events are processed.
2. **Bus.** Allows bus transaction requests to be added, tracks current bus transactions, and selects the next transaction to process.
3. **Core.** Tracks the current operation being performed and gives the next operation.
4. **Cache.** Maintains cache data structures.
5. **CacheController.** Interface for a cache controller that receives core actions and snooped bus transactions stimulus. This is an abstract class that the specific cache coherence protocols have to inherit.

The following sections describes these components in detail.

2.2 Simulator

The Simulator continuously processes events in the order of the time in which they occur until there are none. The initial events are the first four core operations. Each event might result in future events being added. The following describes the four events that are defined.

1. **CoreOpStart.** This represents the start of a core operation. The cache controllers will receive this as a stimulus if it is a memory operation. The exception is that if the cache set is full, the Simulator will perform the eviction first, and maybe a write back to memory, before giving the controller the operation as a stimulus.

2. **CoreOpEnd.** This represents the end of a core operation.

3. **BusRequest.** This represents the start of a bus transaction, and the cache controllers may receive this as a snooped bus transaction stimulus.

4. **BusResponse.** This represents the end of a bus transaction.

Notably, a bus transaction that involves a memory operation will occupy the bus for that cache block for the whole duration to simplify implementation. Therefore, there is no need for a memory object, as the Simulator can just determine when a bus transaction will end, sometimes based on the controllers' output upon snooping the transaction, and create the BusResponse event.

At the end, the Simulator will print the output statistics of the simulation.

2.3 Bus

A key design decision I had to make was to decide between an atomic bus and a split-transaction bus [1]. An atomic bus would lock the bus for the whole duration of the bus transaction, and it will result in a lot of idle time if there is a memory access. However, a split-transaction bus where there are separate lines for requests and responses and concurrent bus transactions for different blocks would be too complicated to implement and requires intermediate states to be introduced in our cache coherence protocols.

Therefore, I settled for an intermediary between the two where the Bus is atomic with respect to a single cache block, but it allows for bus transactions between different blocks to be serviced concurrently. Moreover, there is a limit of four concurrent bus transactions to better emulate hardware constraints. Another key assumption made is that bus traffic that does not include data transfer could travel instantaneously, whereas only data transfers (to-and-fro memory or caches) take time. The shared line can also be asserted instantaneously.

The Bus object maintains queues of bus transaction requests for each cache block and the concurrent transactions that are currently being serviced using hash maps. It allows current bus transactions to be marked as completed and chooses the next bus transaction to service on a first come first serve policy. The Simulator will check the Bus for these next transactions to service and creates the bus events.

2.4 Core

The Core contains the list of operations to perform for that core. It tracks the current operation and gives the next operation when the current operation is marked as completed.

2.5 Cache

A Cache maintains the data structures for an L1 data cache. It uses an array to represent the cache sets. In each cache set, we have a hash map of the blocks in the set to whether it's dirty, and a linked list that maintains the order in which the blocks are accessed - to implement the LRU replacement policy. Methods are provided for block read, write, insertion, and removal, and retrieving which block is to be evicted when the cache set is full.

2.6 CacheController

The CacheController is an abstract class for cache controllers that implement one of the cache coherence protocols. It contains interface methods for receiving core operations stimulus, snooped bus transactions stimulus, and bus response stimulus. The concrete CacheController to instantiate will depend on the "protocol" input for the simulator. The Simulator will use the CacheControllers instantiated to process the events - the controller will receive the stimulus and produce output (e.g. whether to do a flush to memory).

The CacheController will also have access to the bus for adding new bus transaction requests, and its cache for modifying the cache state.

2.6.1 MESI

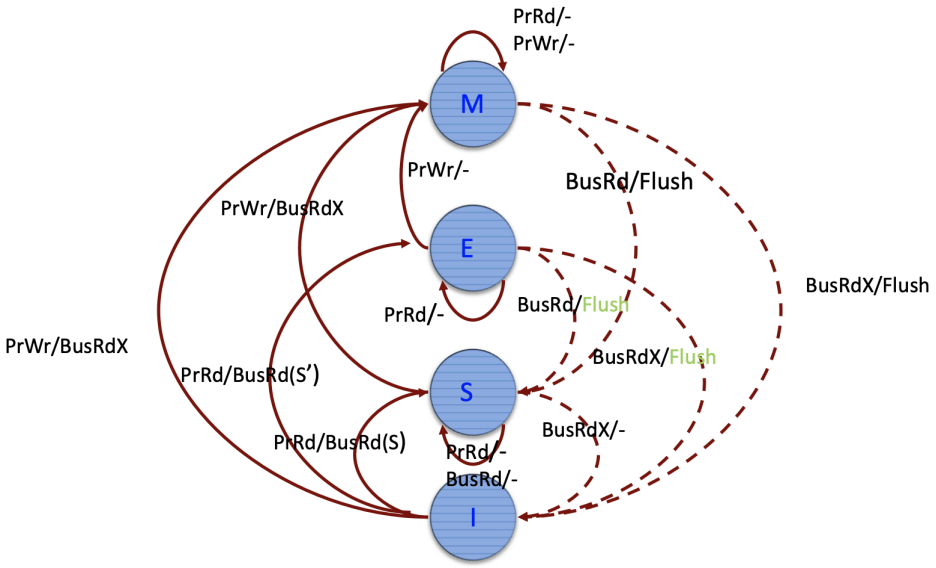


Figure 1: MESI Protocol State Transition Diagram [2]

The MESI protocol implemented is as given in the lecture [2]. The state diagram is taken from the Lecture Notes as shown in Figure 1. The most basic implementation without any cache-to-cache sharing is implemented here, and so the highlighted Flush after a BusRd or BusRdX is snooped in Exclusive state is not performed - the other cache will read from memory instead. Since there is no cache-to-cache sharing, every BusRd or BusRdX will take at least 100 cycles to service as the cache needs to read from memory.

For clarity, the Flush in Figure 1 refers to a Flush to memory. Therefore, if a Flush needs to happen upon a BusRd or BusRdX, it will take an additional 100 cycles. After the Flush happens, the requesting cache can then read from memory, and it will take 200 cycles in total.

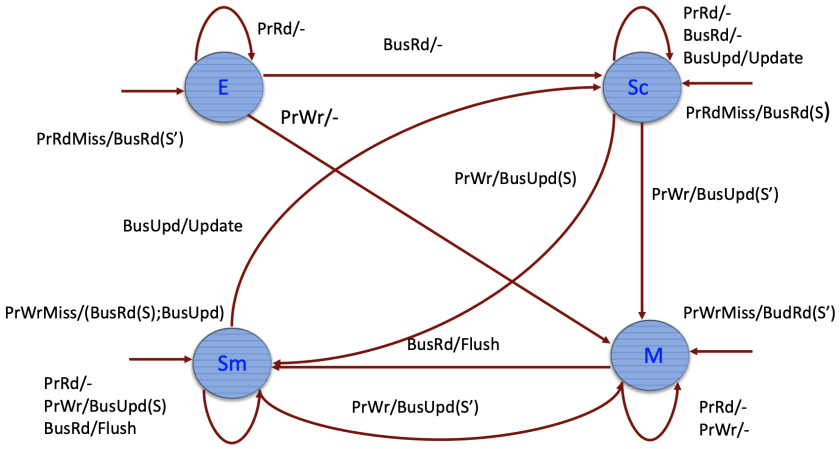


Figure 2: Dragon Protocol State Transition Diagram [2]

2.6.2 Dragon

The Dragon protocol is also implemented as given in the lecture [2] and the state diagram is taken from the Lecture Notes as shown in Figure 2.

A new bus transaction introduced in Dragon is BusUpd, and in our simulation, a BusUpd will take $2N$ cycles to complete, where N is the number of words transferred. An optimization performed in our system is that once the cache controller gets hold of the Bus to perform a BusUpd, but the shared line is not asserted, then the BusUpd data transfer does not happen. For clarity, Update is a local cache update and we assume that it happens instantaneously once the updated block is received from the bus.

2.6.3 Illinois MESI (Advanced Task)

The advanced task implemented is the Illinois MESI protocol [3], which is an improvement over the basic MESI protocol. In the basic MESI protocol, we do not allow any cache-to-cache transfers, and all bus reads have to be serviced by reading from memory. The read from memory is not necessary if another cache already has the block, and a cache-to-cache transfer would take much lesser cycles as compared to reading from memory. The disadvantage is that cache-to-cache transfers require greater hardware complexity.

The following changes are made to the basic MESI protocol to implement Illinois MESI:

1. **Exclusive state.** When a BusRd or BusRdX is snooped in Exclusive state, the cache controller will issue a flush to the requesting cache (cache-to-cache transfer). No write-back to memory is required because the memory is up-to-date. In the basic MESI protocol, no flushes will happen.

2. **Shared state.** Similarly, when a BusRd or BusRdX is snooped in Shared state, the cache controller will also issue a flush to the requesting cache (cache-to-cache transfer). Notably, more than one cache controller might do the flush here, which is a possible disadvantage of the Illinois MESI protocol as it might overload the bus. This disadvantage is not emulated in our software simulation and we assume that the bus will not be overloaded.

3 Methodology

The MESI, Dragon, and Illinois MESI cache coherence protocols were compared using the *blackscholes*, *bodytrack*, and *fluidanimate* benchmarks from the PARSEC benchmark suite.

Metrics used to compare the protocols include the Overall Execution Cycle, total Data Traffic on the Bus, Number of Invalidations/Updates on the Bus, and average Cache Miss Rates.

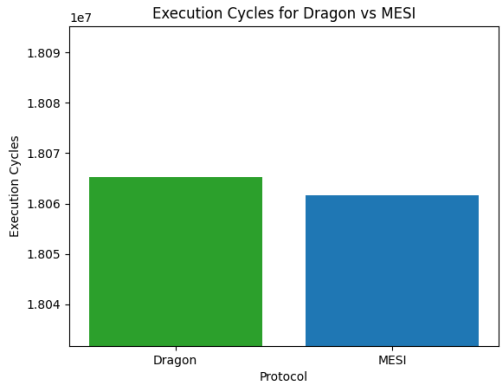
Cache sizes, associativity, and block sizes were also varied to compare the protocols across a range of cache parameters. The following table describes the values of the parameters used:

Parameter	Values
Cache Sizes (bytes)	512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072
Associativity	1, 2, 4, 8, 16, 32, 64, 128
Block Sizes (bytes)	4, 8, 16, 32, 64, 128, 256, 512

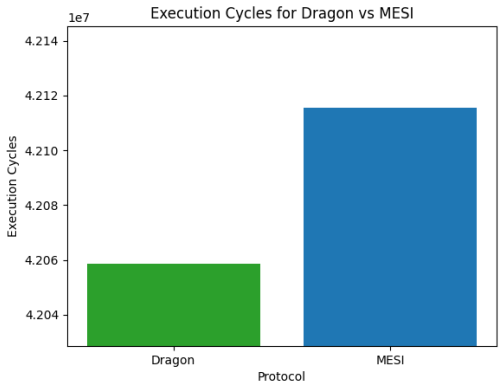
4 Results

This section reports the findings of the experiments. The data for all the experiments can be found in *./experiments/benchmark.csv* in the project folder.

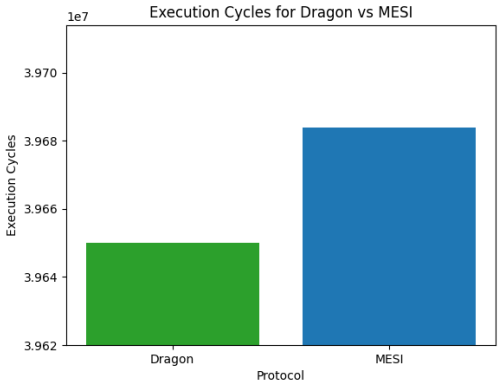
4.1 Dragon vs MESI



(a) blackscholes



(b) bodytrack



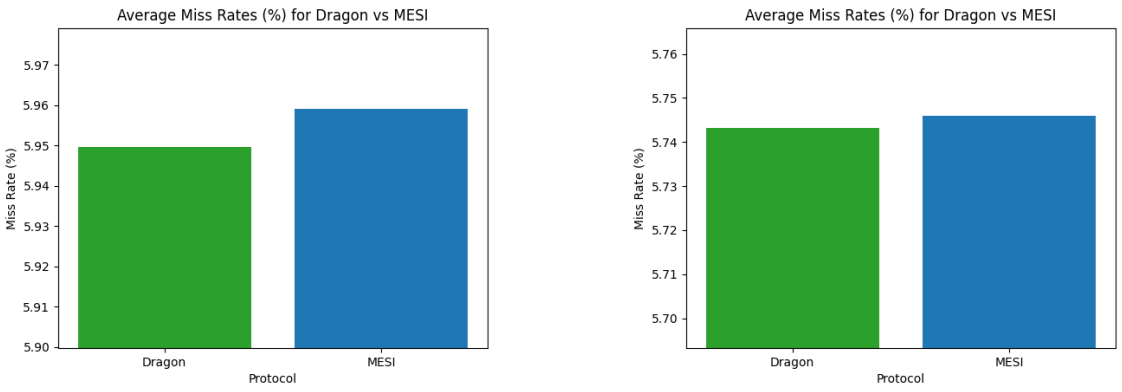
(c) fluidanimate

Figure 3: Execution Cycles of Dragon vs MESI protocol with 4096B Cache Size, 2 Associativity, 32B Block Size

Using the default configuration, the Dragon protocol takes a smaller number of execution cycles to complete for both the bodytrack and fluidanimate benchmarks, but a larger number of

execution cycles for the blackscholes benchmark as shown in Figure 3. This difference is likely due to the differences between the blackscholes benchmark and the other two benchmarks. According to Bienia et al.'s categorization of the PARSEC benchmark suite, the blackscholes benchmark has lower exchange of data between the cores as compared to the other two [4]. This lower exchange of data in blackscholes likely explains Dragon's worse performance in blackscholes as there are a lot of redundant Bus updates issued by the Dragon protocol. On the other hand, when there is greater exchange of data in bodytrack and fluidanimate, the Dragon protocol proves more useful as it updates the other caches instead of invalidating them and minimises the cost of writing to shared data.

In bodytrack, there is also much greater sharing of data between cores as compared to fluidanimate [4]. This likely explains why Dragon's percentage improvement over MESI is even more significant in bodytrack, and why bodytrack takes the greatest number of execution cycles to complete for both the Dragon and MESI protocols as it has high sharing and high exchange.



(a) bodytrack

(b) fluidanimate

Figure 4: Average Cache Miss Rates of Dragon vs MESI protocol with 4096B Cache Size, 2 Associativity, 32B Block Size

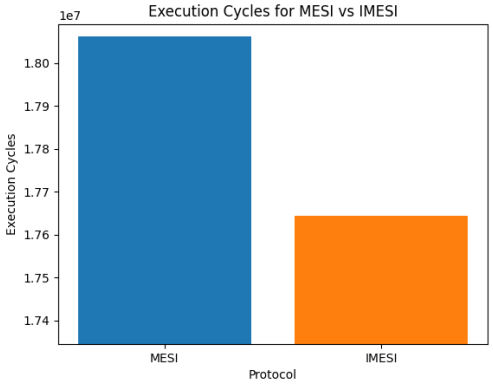
According to Figure 4, the average miss rates across all the cores for the Dragon protocol are also slightly smaller than that for the MESI protocol in both the bodytrack and fluidanimate benchmarks. This again points to the benefit of the Dragon protocol in these two benchmarks as updating the shared cache line instead of invalidating it upon a write allows the other cores to read the cache line with a cache hit instead of a miss.

4.2 MESI vs Illinois MESI

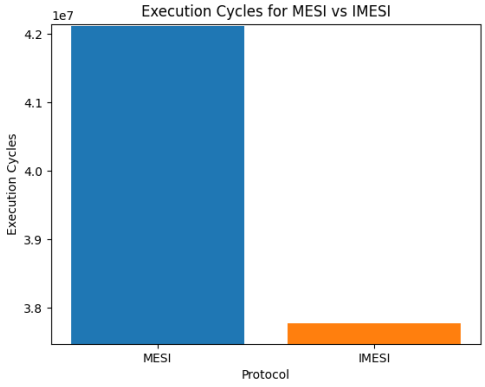
The benefits of the Illinois MESI protocol over the basic MESI protocol are obvious in Figure 5. Illinois MESI gives lower execution cycles across all the benchmarks. This finding is not surprising as the Illinois MESI protocol is the same as the MESI protocol except that there will be cache-to-cache sharing of data whenever possible instead of doing a read from memory, thus saving a lot of redundant memory read cycles.

The benefits of Illinois MESI are most obvious in the bodytrack and fluidanimate benchmarks (with the greatest percentage improvements) as, again, there is more sharing and exchange of data in bodytrack and fluidanimate. The time saving from doing a cache transfer instead of a memory read will thus be more prominent.

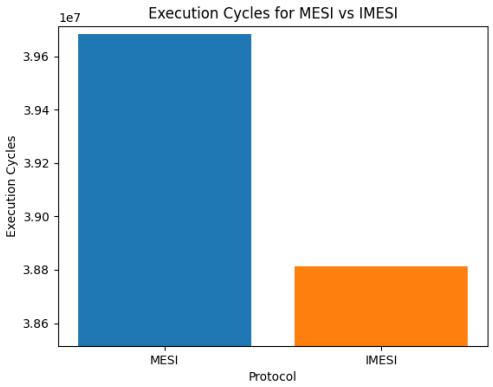
The other metrics between MESI and Illinois MESI are comparable without any significant differences. Therefore, in summary, the main benefit of Illinois MESI protocol over MESI is the time saving from cache-to-cache sharing. The optimization implemented for the MESI protocol in this advanced task thus brings about tangible improvements in terms of execution



(a) blackscholes



(b) bodytrack



(c) fluidanimate

Figure 5: Execution Cycles of MESI vs Illinois MESI protocol with 4096B Cache Size, 2 Associativity, 32B Block Size

time. Notably, the potential downside of Illinois MESI is greater hardware complexity from implementing cache-to-cache sharing. However, this downside is not captured in the Simulator and the metrics measured in the experiments and we should be careful about concluding that Illinois MESI is better in all aspects just based on this set of results.

4.3 Effects of Varying Cache Parameter

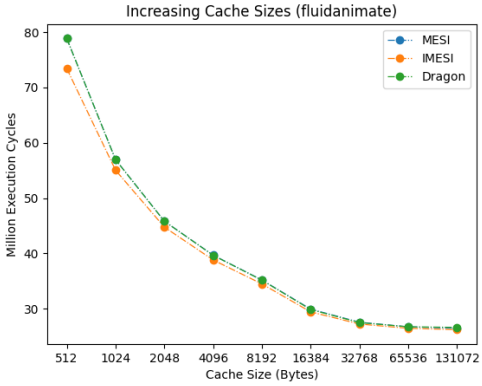
Figure 6 shows the effects on execution cycles for all the protocols when a certain cache parameter is varied while keeping the other parameters constant.

4.3.1 Varying Cache Sizes

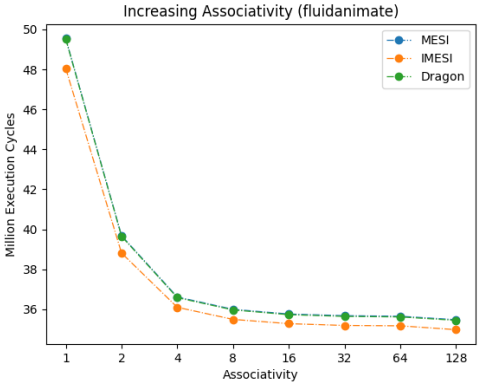
As seen in Figure 6a, increasing cache sizes improve overall execution cycles for all the protocols. All protocols' execution cycles follow roughly the same trend when cache sizes increase, and they hit diminishing returns when the cache size is above 64kB. This improvement is expected as increasing cache size increases the number of cache sets and in turn, lowers cache miss rates.

4.3.2 Varying Associativity

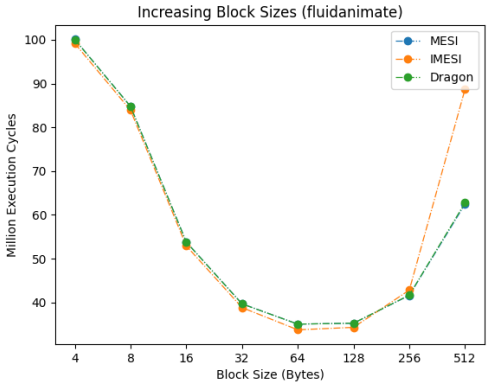
Increasing associativity also brings about lower overall execution cycles for all the protocols, and all of them also follow roughly the same trend when the associativity increases. The increase in associativity likely lowered cache miss rates as evictions are less likely to occur since blocks are able to share the same cache set - this decreases the number of conflict misses. The



(a) Varying Cache Sizes, 2 Associativity, 32B Block Size



(b) 4096B Cache Size, Varying Associativity, 32B Block Size



(c) 4096B Cache Size, 2 Associativity, Varying Block Sizes

Figure 6: Execution Cycles of Protocols for fluidanimate benchmark

improvement in execution cycles also hit diminishing returns when the associativity is above 8. This suggests that we might not want to increase associativity much further if we are obtaining only marginal improvements. This is because greater associativity will lead to greater hardware costs as all the blocks in the cache set have to be searched in parallel when we are checking for a cache hit.

4.3.3 Varying Block Sizes

In contrast, increasing block sizes do not lead to monotonically decreasing overall execution cycles. This is likely because although increasing block size could better exploit spatial locality, it also decreases the number of cache sets and might lead to increased conflict misses. The optimal block size for all 3 protocols is about 64B.

Notably, Illinois MESI seems to do better than both the MESI and Dragon protocols for all block sizes up till 128B. When block size is above 128B, Illinois MESI has a drastic increase in overall execution cycles. This is likely because cache-to-cache transfers time in the Simulator scales according to the block size, and it eventually takes longer than memory access when the block size is too large. This might point to a limitation in the Simulator which has a fixed memory access time of 100 cycles - ideally, memory access time should also scale according to the block size, similar to cache-to-cache transfers.

5 Conclusion

In conclusion, the Cache Coherence Simulator proved useful in helping us compare various cache coherence protocols. Using various benchmarks in our experiments also deepened our understanding of the workload in which a particular cache coherence protocol would be more useful.

The Dragon protocol is more suitable for workloads with greater sharing and exchange of data where there are writes, such as in the bodytrack and fluidanimate PARSEC benchmarks. Moreover, the implementation and evaluation of Illinois MESI also highlighted the tremendous improvements in execution time we can get from cache-to-cache sharing, albeit at the cost of greater hardware complexity.

We also looked at how varying cache parameters could affect overall execution time. Increasing cache size and associativity leads to monotonically decreasing overall execution time but with diminishing returns. We have also determined that the optimal block size for all 3 protocols is about 64B, and that we shouldn't increase the block size too much for the Illinois MESI protocol as it greatly increases the cost of cache-to-cache sharing.

References

[1] CMU, "CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012), A More Sophisticated Snooping Multi-Processor Lecture Notes," March 2012.

[2] T. E. Carlson, "CS4223 Multi-core Architectures, Coherence Lecture Notes," September 2022.

[3] MESI protocol, "MESI protocol — Wikipedia, The Free Encyclopedia," 2022.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Association for Computing Machinery, 2008.