NUS

National University of Singapore

CS4224

Assignment 2 part 2

Professor Erik Carlsson

Øystein Weibell: A0310493L

Viktor Lado Naess: A0309887H

1 Intro

In part 1 of the assignment, we implemented a single-threaded, single-core CPU with a cache and DRAM. This implementation was tested against different traces to evaluate its limits and understand the behavior of its cache. Building on this foundation, part 2 explores memory coherence techniques by implementing two cache coherence protocols: MESI and Dragon.

This report compares the performance of these two protocols across a range of cases and analyzes their strengths and weaknesses. Additionally, we examine the implementation of an optimization known as Silent evictions, an adaptation of the MESI protocol designed to reduce coherence overhead and improve performance.

2 Implementation

The simulator and protocols were implemented in Python, leveraging standard libraries such as `argparse`, `enum`, and `collections`. The implementation is organized into two main components:

- **Data Structures:** These include `ENUM`, `CacheBlock`, `DRAM`, and `AddressHandler`, which model cache states, memory blocks, and address manipulations.
- **Functional Components:** These comprise the `Cache`, `Bus`, `CPU`, and `Simulator` classes, which handle the execution of the MESI and Dragon protocols, synchronization, and metric collection.

This modular structure ensures clarity and facilitates testing and extension of the simulator.

3 Data structures

3.1 ENUM

The `ENUM` data structure defines the states and operations required for the cache coherence protocols. For MESI, the states include `MODIFIED`, `EXCLUSIVE`, `SHARED`, and `INVALID`, while the Dragon protocol uses `MODIFIED`, `SHARED-CLEAN`, `SHARED-MODIFIED`, and `EXCLUSIVE`.

Additional `ENUM` types include:

- `Operation`: Defines `LOAD` and `STORE` for read and write operations on the cache.
- `BusTransaction`: Specifies bus transactions such as `BUS_READ`, `BUS_READ_X`, and `BUS_UPDATE`, which dictate how data is shared or modified across caches.

These definitions ensure clarity and consistency in handling protocol-specific states and transactions.

3.2 Cache Block and DRAM

The `CacheBlock` class encapsulates the essential properties of a cache block, serving as a container for data and metadata required for the MESI and Dragon protocols. Each cache block includes the following attributes:

- **Tag:** A unique identifier for the memory block, used to verify whether the block matches a requested address within the cache set.
- **Set Index:** Indicates the cache set to which the block belongs, facilitating efficient block lookup and placement.
- **State:** Represents the current coherence state of the block (e.g., Modified, Shared, Exclusive, or Invalid), as defined by the MESI and Dragon protocols.
- **Dirty:** A boolean flag indicating whether the block has been modified locally but not yet written back to main memory. This might seem verbose since we already track its protocol state. However, the job of this flag is to make the tracking logic of useful versus useless cache block adoptions easier to implement.
- **Address:** The memory address associated with the block, which combines the tag and set index.

The `CacheBlock` implementation ensures accurate tracking of block-level metadata, enabling precise enforcement of protocol rules during cache operations.

DRAM: The `DRAM` class models the main memory with a defined access latency of 100 cycles. It provides two primary operations:

- `Fetch`: Simulates retrieving data blocks from memory.
- `Write Back`: Simulates writing modified data blocks back to memory.

The DRAM implementation ensures realistic modeling of memory behavior, complementing the cache in the simulator's architecture.

3.3 Address Handling

This set of code, unsurprisingly handles address manipulation. It does this through the attributes, block_size, num_sets, and bit calculations (block_offset_bits, index_bits, and tag_bits) allowing it to parse 32-bit addresses into tag, index, and offset components.

Additionally, it implements the methods:parse_address and get_block_address.

4 Main Classes and Functions

4.1 Cache

The `Cache` class manages cache hits, misses, and state transitions for both the MESI and Dragon protocols. Its responsibilities include:

- **Bus Requests:** Tracks unresolved requests initiated by the cache.
- **State Transitions:** Updates block states based on the coherence protocol and bus transactions.

- **Snooping:** Monitors bus transactions to adjust cache states accordingly.

The cache is implemented as a dictionary, where each key represents a set containing a list of blocks. This structure supports flexible associativity and efficient state management.

4.2 Bus

The `Bus` class coordinates inter-cache communication and handles transactions such as updates, invalidations, and write-backs. Key features include:

- **Transaction Queue:** Manages bus requests in a first-come-first-serve manner.
- **Transfer Cycles:** Simulates transaction latencies based on data size and protocol requirements.
- **Cache List:** Maintains references to all connected caches for broadcast operations.

This class ensures protocol rules are enforced and inter-cache communication is synchronized accurately.

4.3 CPU

The `CPU` class is responsible for processing the instruction trace files and executing operations based on the MESI and Dragon cache coherence protocols. It simulates the behavior of a processor core, handling both memory and non-memory instructions. Key attributes of the `CPU` class include:

- **Instruction Pointer:** Tracks the current position in the trace file, ensuring that instructions are processed sequentially.
- **Compute Cycles Remaining:** Counts down the number of cycles required to complete non-memory instructions, such as arithmetic operations, before proceeding to the next instruction.

The `CPU` class interacts closely with the `Cache` class to ensure correct execution. It processes three types of instructions:

- **Load (Read):** Requests data from the cache or memory, potentially triggering coherence actions.
- **Store (Write):** Updates data in the cache and propagates changes based on protocol requirements.
- **Compute:** Handles non-memory operations, incrementing the CPU's cycle count accordingly.

By accurately modeling instruction execution and waiting states, the `CPU` class ensures realistic simulation of processor behavior in a multi-core environment.

4.4 Simulator

The `Simulator` class orchestrates the execution of cache coherence protocols by:

- **Initialization:** Setting up the caches, bus, and DRAM.
- **Synchronization:** Scheduling instructions and managing cycle-by-cycle execution across cores.
- **Metric Collection:** Gathering performance data, including cache hits, misses, and bus traffic.

The simulator processes memory operations, compute cycles, and bus transactions in a coordinated manner, ensuring accurate timing and protocol adherence.

5 Advanced literature review

While in the end we chose to implement a modification of silent eviction, an explanation of silent evictions can still be of interest as to see the source of inspiration.

Silent evictions is in short an optimization that allows certain cache lines to be evicted without sending notifications to other caches or to main memory. The special cases are predominantly for when cache lines in states where coherence can be maintained without extra messaging, especially for the shared (S) and Exclusive (E) states. It is an add on to the MESI. The key distinction lies in the eviction process.

When an eviction/replacement in the cache occurs when we are in the shared and exclusive state, normally, the MESI protocol would broadcast this to the other caches, for Silent evictions however this is not the case. In this case, the broadcasts are skipped, hence the *silent* in the name. For example in the shared state, the line of interest is known to present multiple caches. When we silently then evict said line, the evicting cache will no longer have it but the other caches (or memory) will and can still respond to future requests for that line.

Or for instance, in the exclusive state, the line is unique to the cache but unmodified. The cache can silently evict the line from the cache, dropping the exclusive copy; however, this is on the condition that said line is identical to the memory content. Then future requests can retrieve the data from memory without issue.

The gain to this method is less communication between the caches. In short, because no other caches rely on the evicting cache for the latest data, there's no need to update or invalidate other caches, allowing the line to be removed with no inter_cache coherence messages. This then leads to *Reduced Coherence Traffic*, *Lower Latency* and better scalability.

Ultimately, silent evictions became a source of inspiration for what we call cache line adoptions. In short, cache line adoptions has the purpose to minimize the penalty of writing back evicted cache lines to main memory, and achieves this by checking the if any other cache can adopt the to be evicted line. And so if possible the system avoids the 100-cycle write-back to memory penalty. We determine that a line can be adopted if by another cache if it exists in the Shared or Exclusive state. We compare the performance improvement between the two.

6 Results and analysis

6.1 MESI vs Dragon

Throughout the tests it is clear that both the MESI and the Dragon have their own respective merits. The following discord, will go through each protocol and check which is best.

6.1.1 Blackscholes

Table 1: Comparison of Dragon and MESI Protocols on Blackscholes with no adoption

Metric	**Dragon Protocol**	**MESI Protocol**
Overall Execution Cycles	47,510,346	13,367,186
Total Data Traffic on Bus	93,916,032 bytes	602,496 bytes
Number of Updates/Invalidations	2,923,668 updates	3,237 invalidations
Average Cache Hit Rate	99.85%	99.84%

Analyzing the results, MESI demonstrates a significant performance advantage in the Blackscholes benchmark. With approximately 28% of the execution cycles required by Dragon and over 99% reduction in bus traffic, MESI's invalidation mechanism minimizes idle cycles and execution delays. This makes MESI well-suited for compute-intensive workloads with frequent shared data access.

Blackscholes is computationally heavy and has a high frequency of shared data accesses. This can provide insight into the difference of performance, where MESI's invalid mechanism reduces idle cycles and execution time in compute-intensive workloads with high shared data access, Dragon's update mechanism introduces significant delays due to high bus traffic and idle times, making it less efficient for the Blackscholes benchmark.

6.1.2 Bodytrack

Table 2: Bodytrack Benchmark Comparison: Dragon vs. MESI Protocols

Metric	**Dragon Protocol**	**MESI Protocol**
Overall Execution Cycles	24,763,728	23,429,146
Total Data Traffic on Bus	9,370,272 bytes	4,700,128 bytes
Number of Updates/Invalidations	183,819 updates	12,280 invalidations
Average Cache Hit Rate	98.63%	98.55%

In the Bodytrack trace, MESI completes the benchmark approximately 5% faster than Dragon. MESI also reduces bus traffic by about 50% compared to Dragon, while the cache hit rates remain similar, with Dragon having a slight edge. However, the lower bus traffic in MESI offsets this minor difference in cache hit rates, highlighting its superior efficiency.

The Bodytrack trace involves a mix of private and shared data accesses. MESI benefits from slightly lower idle cycles, while Dragon incurs higher idle cycles due to the overhead of updating shared data across caches. These delays are a consequence of Dragon's update mechanism, which generates significant bus traffic. Ultimately, MESI outperforms Dragon due to its more efficient management of idle cycles and bus traffic.

6.1.3 Fluidanimate

Table 3: Fluidanimate Benchmark Comparison: Dragon vs. MESI Protocols

Metric	**Dragon Protocol**	**MESI Protocol**
Overall Execution Cycles	15,893,932	15,171,741
Total Data Traffic on Bus	7,262,176 bytes	2,218,048 bytes
Number of Updates/Invalidations	183,078 updates	27,280 invalidations
Average Cache Hit Rate	99.63%	99.47%

In the Fluidanimate trace, MESI demonstrates better efficiency with lower execution cycles and reduced bus traffic. Specifically, MESI completes the benchmark in 15,171,741 cycles, a 4.5% decrease compared to Dragon's 15,893,932 cycles. Additionally, MESI reduces bus traffic by approximately 70%. However, MESI records a slightly lower average cache hit rate than Dragon.

Fluidanimate involves frequent reads and writes to shared data, which significantly impacts protocol performance. Dragon's update mechanism increases bus traffic and idle cycles due to continuous updates of shared data across caches. In contrast, MESI shows higher private accesses, as data often becomes exclusive to processors following invalidations. These characteristics allow MESI to outperform Dragon in Fluidanimate by minimizing bus traffic and idle cycles.

7 Adoption Optimization

The impact of adoption on the Dragon and MESI protocols varies significantly across the three benchmarks, as shown in Tables 4–6. While MESI consistently benefits from adoption, Dragon demonstrates inconsistent results due to its bus-intensive design and adoption management overhead.

For the Dragon protocol, adoption sometimes improves overall execution cycles, such as in the Bodytrack benchmark. However, this improvement is not guaranteed. The added complexity and overhead of managing adoptions in Dragon can offset its potential benefits, especially in workloads with frequent shared data updates. This suggests that the decision to use adoption in Dragon should depend on the shared data access patterns of the workload.

In contrast, the MESI protocol shows more consistent improvements with adoption. It reduces overall execution cycles by minimizing invalidations and optimizing state transitions. Adoptions also reduce idle cycle counts by mitigating delays associated with coherence eviction actions. The underlying invalidation-based mechanism of MESI allows adoptions to minimize bus traffic costs and improve performance. Furthermore, adoptions in MESI enable transitions to the Exclusive state without invalidations, further boosting efficiency.

To better understand the behavior of adoptions, we analyzed the number of "useless adoptions," where adoption adds overhead without tangible performance benefits (see Table 4). Key takeaways include:

- **Blackscholes:** MESI's fewer useless adoptions reflect its efficiency in managing predictable access patterns, outperforming Dragon's update-based approach.
- **Bodytrack:** Dragon's high number of useless adoptions highlights its inefficiency in handling workloads with frequent shared data modifications.
- **Fluidanimate:** Dragon performs better than MESI, showcasing its strength in managing workloads with frequent shared data updates.

These results underscore the importance of workload characteristics in determining the efficacy of adoption for each protocol.

Table 4: Average Useless Adoptions Per Trace and Per Protocol

Benchmark	**Dragon Protocol (Avg)**	**MESI Protocol (Avg)**
Blackscholes	17.5	12.25
Bodytrack	4638.5	74.75
Fluidanimate	134	207

The key takeaways here for the respective files are as follows. For Blackscholes, we see how MESI's fewer useless adoptions supports the conclusion. Its invalidation-based approach's efficiency in managing computational workloads with predictable access patterns being more apt than that of the Dragon protocol. Similarly, Dragon's high level of useless adoptions for the Bodytrack trace highlights its inefficiency in handling workloads with frequent shared data modifications. Conversely, the Fluidanimate traces exhibits the Dragon Protocol doing better, than that of the MESI, reflecting its strength in managing workloads with frequent shared data updates

8 Conclusion

In summary, the MESI protocol consistently outperforms Dragon across all benchmarks in terms of execution cycles and bus traffic. MESI's invalidation-based coherence mechanism reduces communication overhead, making it more scalable for various workloads. While cache hit rates are similar, MESI's efficiency stems from its ability to handle shared data with minimal latency and traffic.

The adoption mechanism synergizes effectively with MESI, enabling transitions to the exclusive state without invalidations. This reduces idle cycles and improves overall performance. However, for Dragon, adoption's benefits are inconsistent due to the protocol's bus-intensive design, which can offset potential gains with additional overhead.

Overall, MESI emerges as the more robust and scalable protocol, particularly when paired with optimizations like adoption.

Table 1: Performance Metrics Comparison for Blackscholes Benchmark

Protocol	**Adoption**	**Overall Execution Cycles**	**Total Bus Traffic (bytes)**	**Updates/Invalidations**	**Idle Cycles (CPU 0)**	**Cache Hit Rate (Avg)**
Dragon	Without	47,510,346	93,916,032	2,923,668 Updates	28,562,357	99.85%
Dragon	With	48,196,970	95,294,208	2,970,075 Updates	29,397,663	99.87%
MESI	Without	13,367,186	602,496	3,237 Invalidations	2,784,470	99.84%
MESI	With	13,255,686	521,152	3,235 Invalidations	2,624,136	99.87%

Table 2: Performance Metrics Comparison for Bodytrack Benchmark

Protocol	**Adoption**	**Overall Execution Cycles**	**Total Bus Traffic (bytes)**	**Updates/Invalidations**	**Idle Cycles (CPU 0)**	**Cache Hit Rate (Avg)**
Dragon	Without	24,763,728	9,370,272	183,819 Updates	7,034,473	98.63%
Dragon	With	25,068,891	8,593,760	155,152 Updates	7,339,636	98.67%
MESI	Without	23,429,146	4,700,128	12,280 Invalidations	5,699,891	98.55%
MESI	With	23,398,008	4,622,240	11,661 Invalidations	5,668,753	98.60%

Table 3: Performance Metrics Comparison for Fluidanimate Benchmark

Protocol	**Adoption**	**Overall Execution Cycles**	**Total Bus Traffic (bytes)**	**Updates/Invalidations**	**Idle Cycles (CPU 0)**	**Cache Hit Rate (Avg)**
Dragon	Without	15,893,932	7,262,176	183,078 Updates	4,385,000	99.63%
Dragon	With	16,084,744	7,337,312	187,407 Updates	4,489,885	99.64%
MESI	Without	15,171,741	2,218,048	27,280 Invalidations	3,630,129	99.48%
MESI	With	15,238,394	2,312,480	30,382 Invalidations	3,807,554	99.48%