Cache Coherence Simulator

Axel Lundberg Lukas Doellerer

Contents

1 Introduction

A cache simulator mimics the behavior of a real caching system. While the internal operations might differ substantially from the, often partially in hardware implemented, cache system, the simulator produces the same result and can therefore be used to evaluate certain design decisions without building a complex, production ready cache.

Simulating only single threaded systems would limit the usefulness drastically, which is why an advanced simulator can reproduce the behavior of caches in systems with an arbitrary amount of cores and therefore caches. Special protocols are used for cores with shared memory in the communication between the caches to keep the cached data coherent for all cores. This means specifically that two caches must not store different data for the same memory address.

Two cache coherence protocols are used and evaluated in the simulator whose implementation is described in this report: MESI and Dragon. MESI is an invalidation-based coherence protocol that is commonly used for write-back caches. The Dragon protocol on the other hand is an update-based cache coherence protocol that does not invalidate cache lines but update them with the new values directly.

For simulating these protocols, the underlying system can be abstracted so far that there is no need to actually execute a program to simulate caching behavior. Our simulator uses traces containing entries of the following form:

```
<Label> <Address> ... <Label> <Address>
```

A label can have the values 0 (load), 1 (store) and 2 (other). The load and store are memory operations that require the cache. The "other" instruction bundles all instructions that do not access the memory. In this simulator, such instructions will be interpreted as a stall in each core for the value specified in the address field. Each core that executes the program generates one such trace. The traces we use to test our simulator are generated from the following benchmarks from the PARSEC suite:

1. **blackscholes**. Option pricing with the Black-Scholes Differential Equation.
2. **bodytrack**. Body tracking of a person.
3. **fluidanimate**. Fluid dynamics for animation purposes using the Smoothed Particle Hydrodynamics (SPH) method.

Each benchmark contains traces for four different cores. Because the simulator is concerned about simulating the *behavior* of each cache with respect to some cache coherence protocol, actual data is unimportant. The simulator is only cares about the cache lines the data would occupy, hence only the address of each memory operation is important.

The implementation compiles to an executable called `coherence` with a couple of input parameters. Listing 1 shows how to execute the binary:

```
coherence <PROTOCOL> <INPUT_FILE> <CACHE_SIZE> <ASSOCIATIVITY> <BLOCK_SIZE>
```

The cache size and block size are specified in bytes. The default configuration is shown in Listing 1.

```
coherence <PROTOCOL> <INPUT_FILE> 4096 2 32
```

The traces from the benchmarks contain a lot of instructions and may take a couple of minutes to run. We had access to a server that can assist us in running the benchmarks. Because each simulation still took quite some time, we started with running the default configuration for each benchmark and protocol. Whichever configuration had the best performance was our baseline where we then optimized the cache size, cache associativity and the block size.

In addition to measuring the performance of the different benchmarks with respect to MESI and Dragon, an improved version of the MESI protocol was also used in the benchmarks. More about the improvement is written in Section 3. This was an advanced task beyond the scope of getting the cache coherence simulator in place.

To be able to measure the performance of our simulator for each benchmark, we implemented a logging mechanism. After each run, the following statistics are collected and emitted to the systems standard output:

1. Overall execution cycles
2. Distribution of private and shared data accesses
3. Number of cache hits and misses
4. Overall bus traffic in bytes
5. Number of invalidations (MESI) or updates (Dragon) sent via the bus
6. For each core:
    (a) Number of executed instructions
    (b) Number of cycles the core took to execute all its instructions
    (c) Number of cycles spent processing “other” instructions (compute cycles).
    (d) Number of idle cycles (cycles the core has to wait in order for the cache to complete its operations)
    (e) Number of memory instruction, also divided into load and store instructions.
    (f) Cache hit and miss rate

1.1 Assumptions

In order to derive a clear specification of the behavior of the cache coherence simulator, a couple of assumptions need to be made. Therefore, the assumptions stated in the project description have been expanded with additional assumptions which remove any undefined behavior.

The project description lists the following assumptions to specify the core behavior of the simulator:

1. Memory addresses are 32-bit wide.
2. The word size is 4 bytes.
3. A memory reference points to 32-bit (1 word) of data in memory.
4. Only the data cache will be modeled.
5. Each processor has its own L1 data cache.
6. The L1 data cache uses a write-back, write-allocate policy and an LRU replacement policy.
7. The L1 data caches are kept coherent using a cache coherence protocol.

8. All the caches are empty on the start of the simulation.
9. The bus uses the first come first serve (FCFS) arbitration policy when multiple processors attempt to schedule bus transactions simultaneously. Ties are broken arbitrarily.
10. The L1 data caches are backed up by main memory — there is no L2 data cache.
11. An 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.

The following assumptions have been added to further specify the behavior of our simulator:

1. Instruction scheduling happens instantly. This means that scheduling an “other”-instruction and executing it for the first cycles happens in the same cycle.
2. Writing to an addresses always takes at least one cycle to hit the cache (write-allocate policy). This means that a write hit incurs a delay of one cycle, a write miss the delay of one cycle with the additional cache miss penalty.
3. A bus update always transmits a single word (32-bit), bus flushes always transmit the full cache line (one block).
4. A bus flush always updates the corresponding block in main memory and therefore requires at least 100 cycles. Updates can target other caches only, making them faster with a minimum time of 2 cycles.
5. The time a scheduled bus transaction takes is counted beginning in the clock cycle *after* the task was put on the bus. This means that a write-back requires a total of 101 cycles until the next action can be performed. This delay consists of one cycle to schedule the write-back flush transaction and 100 cycles for the bus to finish the flush transaction to main memory.
6. Caches block during their own bus transactions. The cache waits for its own bus transaction to finish before it commences finishing the current instruction. This behavior makes it simple to restart the transaction in case it cannot be executed successfully.
7. Other caches may listen to the bus during flushes to main memory and can therefore directly update their stored value. This means that a bus read that causes a flush (for the MESI protocol) only takes the time that is required to flush to main memory (which is greater than shared read time).
8. A *cache hit* occurs if the requested address lies in a block that is currently stored in the cache. It is not affected by the protocol’s state of the line. This means that accesses to invalidated cache lines are also counted as cache hits, even though they incur bus transactions to read the corresponding cache line.
9. Special assumptions for operation under the Dragon cache coherence protocol:
    - For the Dragon protocol, bus flushes are only required for write-backs (elimination of an owned cache block). As long as the copy stays in the cache, every “flush” in the original state transition diagram of the Dragon protocol is replaced with a bus update transaction. No data is written to main memory.
    - Replacements of blocks that are in *Shared-Clean* (Sc) state are not broadcast on the bus.
    - All cache line states are eligible for cache-to-cache data sharing. This means that reads from memory are only required if none of the other caches currently holds the requested address.

- Processor writes schedule a bus update transaction of the affected cache block is in Sc or Sm state. If no other cache responds to the update, then the bus is cleared in the same cycle. This way, the cache can check if other caches still hold the value and if not, only block the bus for one cycle.

1.2 Methodology

The cache coherence simulator is implemented using the programming language *Rust.* Rust is a compiled language with a performance similar to that of C and C++. The Rust compiler ensures strict invariants and induces a certain coding style. This additional effort results in safer code by preventing common bugs and mistakes. While this means that more time has to be spent on getting a prototype working, it reduces the time spent on debugging.

Alongside with the cache coherence simulator, unit tests and integration tests are conducted to assert the correct behavior of the simulator.

2 Method

The following section describes how the different components of the simulator work. It also includes a more detailed description of the the MESI and Dragon protocols.

2.1 Components

Figure 1 shows an overview of how the different components in the simulation are structured. A loader component unpacks a zip archive and passes each file as a *record* stream to a corresponding core in the system. Each core has one L1 cache which is connected to the central, shared bus.

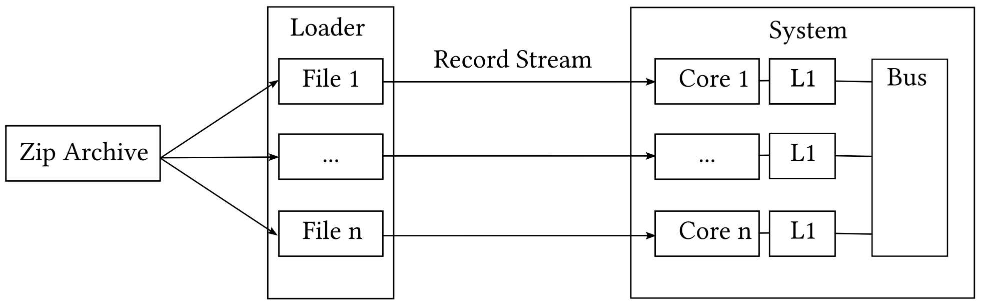


Figure 1: Overview

2.1.1 Record

A record consists of a label and a value. During the initialization of the simulation, the loader decompresses and unpacks the input zip archive and converts each contained file to a stream of records. Given the following line in one such an input file

```
0 0x817ae8
```

a record will be created in the following form:

```
Record {
    label: Label::Load,
    value: 0x817ae8
}
```

2.1.2 System

The system maintains the overall state of all the cores and updates the cores' internal states every cycles. During creation, the system initializes a new core for each supplied record stream.

Updating the cores' internal state consists of three distinct stages: `step`, `snoop` and `after_snoop`. The execution of these stages is synchronized between cores, meaning that all cores have to be finished with one stage until any core can enter the next stage.

The `step` stage lets the core parse the next instruction from the record steam (if the core is currently not busy) and updates the core's cache's internal state appropriately, without any information of the other caches. This means that e.g. all bus reads are expected to result in the "exclusive" state.

The `snoop` stage lets the core's cache snoop on the bus. If the bus currently has an active task from one of the other cores, the cache may update its internal state given the active task and current state of the affected cache line. This stage is responsible for altering the current bus transaction if e.g. the requested address is shared, to inform the requesting cache of its state.

The `after_snoop` stage is a cleanup stage that is executed after the snooping stage. It is required, because caches might have assumed wrong conditions during the initial `step` phase. These caches may use this phase to update their internal state based on their (now altered) active bus transaction. This stage e.g. turns a previously wrongly assumed "exclusive" state to the "shared" state (MESI) if the bus transaction was changed by another core's cache to be a shared read.

2.1.3 Core

The core maintains the state of its L1 cache and its record stream.

2.1.4 Cache

The cache stores the state of each cache line, the corresponding LRU values, which protocol is in use, as well as the address layout of the cache. This layout is computed during initialization of the cache. It uses the block-offset length, the index length, the tag length, the cache-set size and the block size to provide methods for calculating the tag and index of a given address. It is also used to convert between two cache representations, where one is made of nested vectors that follow the idea of a cache that contains cache sets that contain the cache lines. The other representation that we use is similar to the nested version, but flattened into a single vector of contiguous elements.

The cache lines and LRU are both stored as a two dimensional vector containing an unsigned integer, where the rows represent the sets and the columns represent the cache lines. Note that there is no need to actually index into the cache lines using the block offset, because we only simulate the cache without actually storing the cached values.

During the `step` phase (2.1.2) of a core, the cache's update function is called, if the core is not currently stalled by an "other"-record. This update function first checks if the bus is currently processing a transaction that belongs to this cache. The cache will return immediately if this is the case. The cache therefore blocks further execution of new instructions until its last transaction is finished. If this is not the case, the cache starts to execute the next scheduled load or store instruction, if there is any. To execute a load, the cache searches for the given address to see if it is already present in the cache (hit). In case of a cache miss, it takes the necessary steps to load the cache block from main memory or one of the other caches. If the new cache block's cache set is already full, a write-back of the evicted cache block is commissioned. A store instruction also causes the cache to search for the given address to see if it present in the cache. If not, a read or read-exclusive (depending on the protocol) is issued, because of the write-allocate policy. Any required bus transactions are either directly put on the bus or delayed if the bus is currently already used by another cache's transaction.

2.1.5 Bus

The bus is central storage object that contains one storage slot for a bus *task*. This limits the bus to only work on one task at one time. A task has the following form:

```
Task {
    issuer_id: usize,
    remaining_cycles: usize,
    action: BusAction,
}
```

The task contains the ID of the core whose cache issued the task, the remaining clock cycles until the task is finished and the type of bus action. There are a couple of bus actions which are shared between both protocols, however a bus action like `BusUpdShared` is only valid for the Dragon protocol. These protocol specific actions are only issued by the corresponding protocol and are otherwise ignored. The bus actions are represented in the following form:

```
BusAction {
    BusRdMem(address, n_bytes),
    BusRdShared(address, n_bytes),
    BusRdXMem(address, n_bytes),
    BusRdXShared(address, n_bytes),
    BusUpdMem(address, n_bytes),
    BusUpdShared(address, n_bytes),
    Flush(address, n_bytes),
}
```

Some of the bus transactions have been specialized into two distinct forms, a memory and a shared form. This is used to indicate whether the read or update only concerns memory (and therefore takes more cycles) or if it goes to one of the other caches, possibly reducing the required time.

Each bus action state contains an address and the number of bytes that should be transmitted. Each bus update cycle (which corresponds to one cycle), the bus reduces the number of remaining cycles of the currently active bus transaction, if there is one.

2.1.6 Protocol

The protocol is implemented as a Rust *trait*, which defines shared behavior. This is very similar to how interfaces work in other languages (e.g. Java), with some minor differences. For example, traits cannot declare fields. The following methods are defined by the protocol trait:

```
read()
```

```
write()
snoop()
after_snoop()
writeback_required()
invalidate()
is_shared()
```

These operations are required for the implementation of the MESI protocol and the Dragon protocol. As briefly described earlier in the cache section, the cache keeps track of the current protocol in use. When the cache is doing operations on specific cache lines, it invokes the proper method for the underlying protocol as defined in the trait. Thus, the cache only stores a reference to a protocol

```
protocol: Box<dyn Protocol>
```

and operations are invoked on this instance:

```
protocol.snoop(...)
```

This is really useful, because the cache does not need to know which protocol is used and can therefore be built more abstract.

2.2 Coherence Protocols

Like described in Section 2.1.6, each protocol needs to implement a set of methods to satisfy the protocol trait. The following two sub sections will go in depth how the MESI protocol and Dragon protocol implements these, as well as show the state diagrams that come up.

2.2.1 MESI

A transition diagram for our implementation of the `step` phase for the MESI protocol can be found in Figure 2. However, this diagram is a bit different from the transition diagram found on Wikipedia [6], because it only describes our first transition phase. The transition diagram found on Wikipedia has a transition from the invalid state (I) to the shared state (S), which cannot be found Figure 2. This among other non-trivial transitions will be described in this section.

$I \longrightarrow S$. The transition from I to S is instead modeled using the step phase (see Figure 2) and the after snoop phase (see Figure 4). When a cache miss occurs, the cache line is moved from invalid (I) to exclusive (E) as seen in Figure 2. This will issue a bus transaction as a miss occurred. Another core that is currently holding the same cache line will snoop the bus and change the transaction to signal that it is currently holding the same cache line. The core that issued the bus transaction will in the after snoop phase acknowledge the shared signal and change its state to S, see the transition from E to S in Figure 4.

$I \longrightarrow E$. When a cache miss occurs, the core will transition to E. It will issue a bus transaction, but since no other core is holding the same cache line, the transaction will not be changed in the snoop phase, eliminating the chance of transitioning to S in the after snoop phase.

$I \longrightarrow M$. When a cache write miss occurs the core must first fetch the data due to write-allocate. A write-allocate is simulated by stalling for the standard penalty amount of a read miss, while transitioning the cache line to state M as seen in Figure 2. A `BusRdXMem` bus transaction will be issued. Other cores that are snooping will notice this. They will invalidate their cache lines and transition to I.

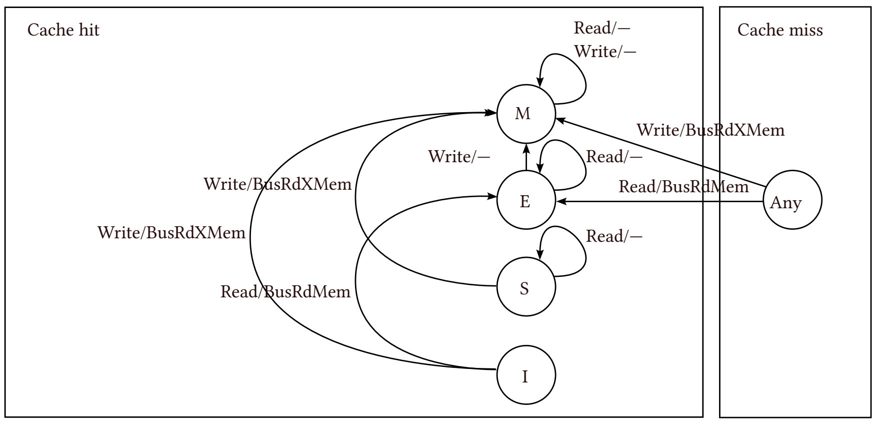


Figure 2: MESI. The step phase of the update cycle.

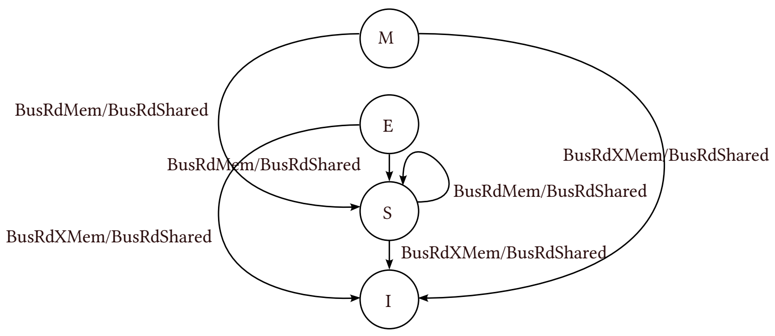


Figure 3: MESI. The snoop phase of the update cycle.

$S \longrightarrow M$. When a cache hit occurs while a core is writing to a cache line that is in S, a transition to M will occur and a `BusRdXMem` bus transaction will be issued. Other cores that are snooping will notice this. They will invalidate their cache lines and transition to I.

$E \longrightarrow M$. When a cache hit occurs while a core is writing to a cache line that is in E, a transition to M will occur but no bus transaction since the core can be sure that has the only copy of the cache line.

From a bus transaction, a cache line can either go to S or I by looking at the Figure 3. When a `BusRdMem` occurs the cache line will go to state S, as it knows that some other core has currently accessed the same cache line. This will as mentioned happen in the snoop phase. The core that is changing the cache line to S will also change the type of the bus transaction to allow the scenario described for transition ($I \longrightarrow S$). Moreover, a core that is reading a cache line that another core has in its cache should be able to transition to S in the after snoop phase, while the other core should be able to switch to S in the snoop phase. When a `BusRdXMem` occurs the cache line will go to state I, as it knows that some other core has currently written to the same cache line. A cache line can also go to state I if it is evicted. When a cache line in M is evicted or invalidated, it must be flushed. A flush will result in stalling all cores until the

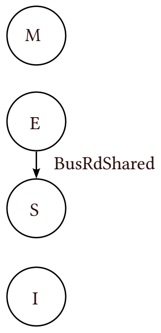


Figure 4: MESI. The after snoop phase of the update cycle.

operation is complete.

2.2.2 Dragon

A transition diagram for our implementation of the processor initiated transitions for the Dragon protocol can be found in Figure 5. Compared to the implementation of the MESI protocol, the implementation of the Dragon is more impacted by the after snoop phase. This is to make sure that the correct transitions occur when dealing with shared cache lines. Since the Dragon protocol has two shared states, the transition diagram is more complicated. Like the MESI protocol, our implementation does not line up directly with the transition diagram shown on Wikipedia [5]. One noticeable difference is that there are no transitions from an “invalid” state to state Sc, Sm and M. Like the previous section all non-trivial transitions will be described in this section. The “Any” state will denote a cache line that is not currently in E, Sc, Sm or M and an access to a cache line in this state is guaranteed to miss.

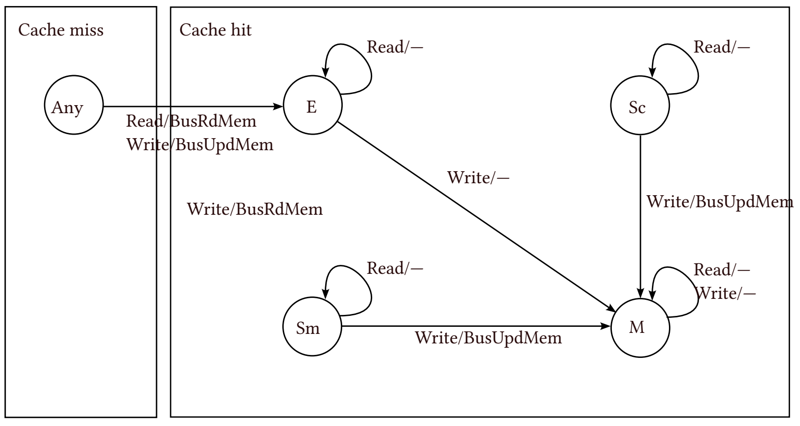


Figure 5: Dragon. The step phase of the update cycle.

Any $\rightarrow$ *E.* When a cache miss occurs while a core is reading to a cache line the cache line will enter state E. A bus transaction will be issued, telling other cores that a read has happened. No other core has a copy of the cache line, so the cache line will remain in state E.

Any ⟶ Sc. When a cache miss occurs while a core is reading to a cache line the cache line will enter state E. A bus transaction will be issued, telling other cores that a read has happened. At least one other core has a copy of the cache line, so these cores will during the snoop phase notice the bus transaction, and update the bus transaction to `BusRdShared`. The core that issued the read, will in the after snoop phase transition from state E to state Sc because of this bus transaction.

Any ⟶ M. When a cache miss occurs while a core is writing to a cache line the cache line will enter state E. A `BusUpdMem` bus transaction will be issued, telling other cores that a write has happened. No other core shares the same cache line, so the cache line will transition to state M in the after snoop phase as the bus transaction does not change, see Figure 7.

Any ⟶ Sm. When a cache miss occurs while a core is writing to a cache line the cache line will enter state E. A `BusUpdMem` bus transaction will be issued, telling other cores that a write has happened. At least one other core has a copy of the cache line. If a core's cache line is currently in either state M or Sm, it will during the snoop phase, transition to state Sc and update the bus transaction to a `BusUpdShared`, see Figure 6. They will have to propagate their data. This is represented as a penalty where all cores are stalled as no data is transmitted. The core that issued the write will transition from state E to state Sm in the after snoop phase, because of the `BusUpdShared`.

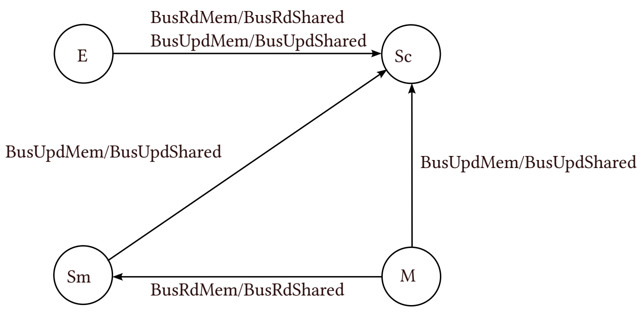


Figure 6: Dragon. The snoop phase of the update cycle.

Sc ⟶ M, Sm ⟶ M. When a cache hit occurs while a core is writing to a cache line that is in state Sc or Sm, the cache line will transition to state M and issue a `BusUpdMem` transaction. No other core is sharing the cache line anymore, so no core will update the bus transaction to `BusUpdShared`. Because of this, no transition will happen in the after snoop phase. The bus will be explicitly cleared since there is no need to update other cache lines.

Sc ⟶ Sm. When a cache hit occurs while a core is writing to a cache line that is in state Sc, the cache line will transition to state M and issue a `BusUpdMem` transaction. Any core that is in state M will transition to state Sc during the snoop phase as a result of the bus transaction. It will update the bus transaction to `BusUpdShared`. This will allow the cache line that issued the write to transition to state Sm in the after snoop phase.

Sm ⟶ Sm When a cache hit occurs while a core is writing to a cache line that is in state Sm, the cache line will transition to state M and issue a `BusUpdMem` transaction. There are at least one other core sharing the same cache line which is guaranteed to be in state Sc. During the snoop phase, this cache line will update the bus transaction to `BusUpdShared`, so that the cache line that issued the write can go back to state Sm.

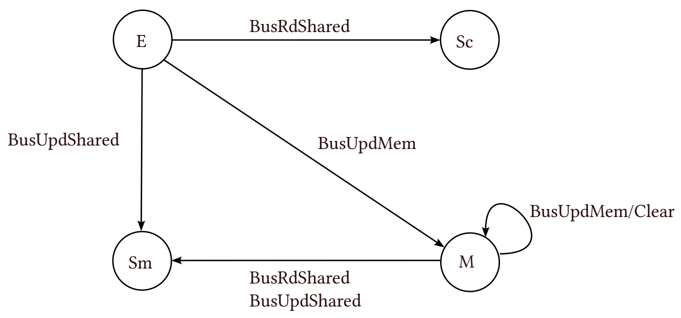


Figure 7: Dragon. The after snoop phase of the update cycle.

The remaining states are rather trivial, e.g. will not issue a bus transaction and the cache line will remain in the same state. Whenever the bus transaction is changed to a shared bus transaction (`BusRdShared` and `BusUpdShared`) the remaining cycles that the cores have to wait will be reduced to symbolize the cache-to-cache transfer.

2.3 Testing

To verify that all state transitions work, a set of unit tests for the MESI protocol and the Dragon protocol has been constructed. The tests are manually created to test every possible state transition of both protocols.

These tests do not test all possible scenarios, though. For example, in the MESI protocol a cache line can transition to state I if it is invalidated or evicted. The tests implemented only invalidate a cache line to make sure that it goes to state I.

Some integration tests of smaller scale have been established as well to verify that all components work as intended. These test only simulate a single-core program. This verifies that the cache implementation is correct, while not necessarily verifying that the cache coherence protocol implementation is correct.

3 Advanced Task

While implementing the two previously described protocols, we quickly noticed that both protocols leave potential for optimizations. Most of their inefficiencies could be removed by better distributing information between the caches and their controllers. This however introduces more bus traffic or more complexity into the processor design. Researchers [2, 3, 4] therefore introduced a new optimization that does not introduce additional bus traffic and is of minor complexity. They proposed a technique called *Read-Broadcast* [1] for snooping and invalidation based cache coherency systems.

Suppose we have a system with multiple cores that each hold the same block in their caches. We also assume that at some point one of the cores initiates a write to this block and therefore sends an invalidation signal to all the other caches. If one of the other cores now were to issue another read to the same block, this read would result in a cache miss because of the previously received invalidation of the cache block. This read miss occurs for every one of the reading cores that got invalidated and all of them would need to re-read the block's value from memory.

In a cache coherency system with the *Read-Broadcast* optimization, all caches snoop on the bus line to detect reads of cache blocks they currently hold. If their stored version is marked as invalid, they replace it with the block that is currently sent over the bus.

We implemented this optimization for our simulator and evaluated its performance improvements in Section 4.2.

4 Results

The following sections will compare the different implementations of the cache coherence protocols. The quantitative analysis section will compare the MESI protocol and the Dragon protocol while the advanced section will evaluate the benefits of the optimization of the MESI protocol.

4.1 Quantitative Analysis

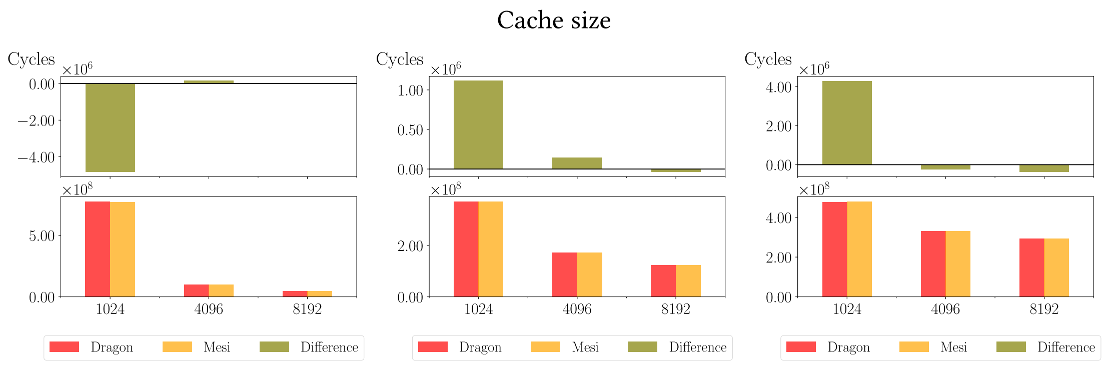
The analysis in this section is based on the default configuration described in the project description. The MESI protocol and the Dragon protocol have been evaluated on every benchmark trace while varying one of either the cache size, the block size or the associativity. Each plot contains two subplots, one that shows the absolute values for both the MESI protocol and the Dragon protocol (the bottom subplot) and one that shows the relative difference of the absolute values of the two protocols (the top subplot), because they are quite similar. This is useful to distinguish the overall pattern when varying a parameter as well as see which of the protocols performed best. The difference is always calculated using

$$P_i^{\text{MESI}} - P_i^{\text{Dragon}}$$

where $P_i^{\text{<protocol>}}$ is the value for each protocol, for each parameter value i in each plot. A negative difference means that the Dragon protocol required more cycles, a positive means that MESI was slower than Dragon. Since it takes quite some time to run each benchmark trace, there will only be two shifts for each parameter. This already constitutes a large number of tests,

$$\underbrace{2}_{\text{2 protocols}} \times \underbrace{3}_{\text{3 traces}} \times \underbrace{2}_{\text{2 shifts}} \times \underbrace{3}_{\text{3 parameters}} + \underbrace{3 \times 2}_{\text{default}} = 42$$

without accounting for the advanced tests. Figure 8 shows the performance for each protocol when varying the cache size.



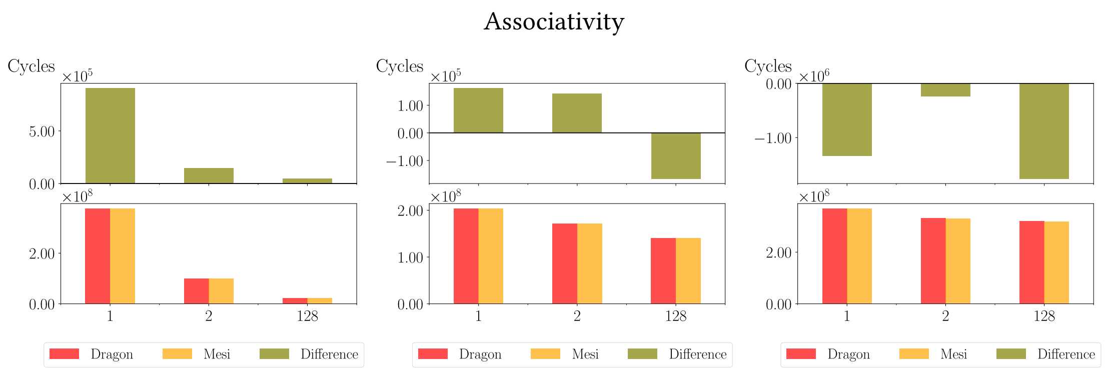
(a) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

Figure 8: The graphs show the number of executed cycles when varying the size of the cache. There are three different settings for the cache size, 1024 bytes, 4096 bytes and 8192 bytes. The associativity is 2 and the block size is 32 bytes.

We can see that the MESI protocol performs better for the Blackscholes benchmark, while the Dragon protocol performs better for the Bodytrack and the Fluidanimate benchmark when the cache size is 1024. However, the difference is only significant for this cache size, and as we increase the cache size, both protocols have more or less the same performance. The overall pattern is that the bigger the cache size is, the fewer clock cycles are required, while the impact of increasing the cache size seems bigger in the Blackscholes. Bigger cache sizes lead to fewer cache misses which will reduce the number of clock cycles. However, there is a tradeoff, since bigger memories have higher access time. The result of the simulator is not affected by this tradeoff.

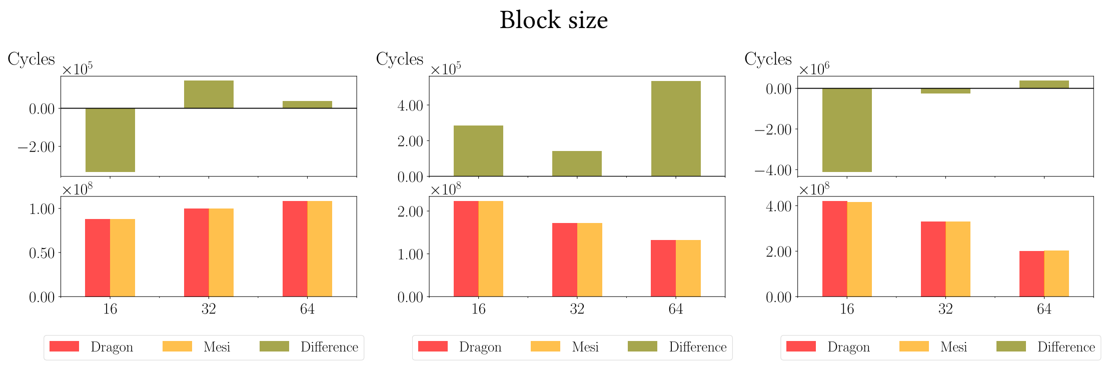
In Figure 9 we can again see that the overall pattern of increasing in this case the associativity decreases the number of clock cycles for each benchmark trace. The impact is again greater in the Blackscholes benchmark. However, the Dragon protocol performs better in the Blackscholes benchmark, and the MESI protocol performs better in the Fluidanimate benchmark. Higher degree of associativity will in principle reduce the number of cache misses. However, there is again a tradeoff since a higher degree of associativity will require more parallel searches for the correct cache line. The result of the simulator is again not affected by this tradeoff.

In Figure 10, the overall pattern is not the same in every subplot. In the Blackscholes benchmark it seems like increasing the block size increases the number of clock cycles. For the Bodytrack benchmark and the Fluidanimate benchmark, increasing the block size decreases the number of clock cycles, indicating that these two benchmarks deal with a lot of consecutive data where it is beneficial to load bigger chunks of data. The MESI protocol performs better in the Blackscholes benchmark and the Fluidanimate benchmark, but the differences are more subtle the higher the block size is.



(a) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

Figure 9: The graphs show the number of executed cycles when varying the associativity. There are three different settings for the associativity, 1 (direct mapped), 2 (2-set-associative) and 128 (fully associative). The block size is 32 bytes and the cache size is 4096 bytes.

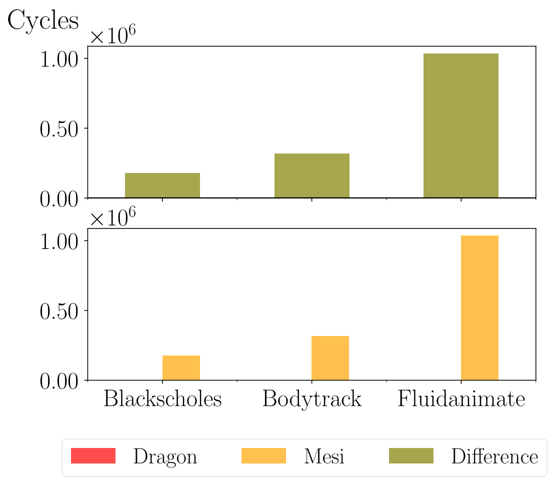


(a) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

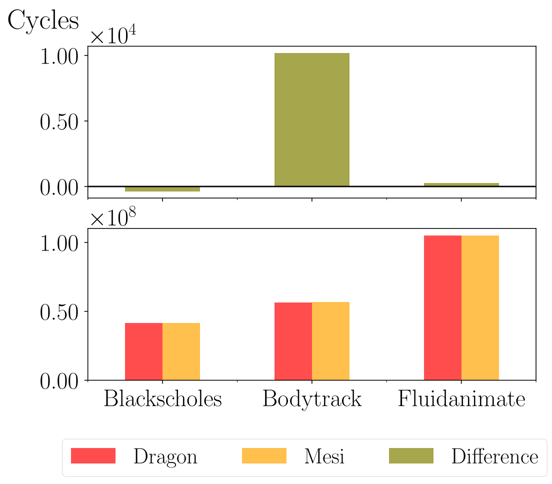
Figure 10: The graphs show the number of executed cycles when varying the block size. There are three different settings for the block size, 16 bytes, 32 bytes and 64 bytes. The associativity is 2 and the cache size is 4096 bytes.

Given the shifts of the cache size, associativity and the block size it is hard to identify which of the protocols is the overall winner. It depends on the benchmark, and the difference is not that great after all if we consider the absolute values for each parameter in each subplot. They have a similar performance.

Looking at Figure 11, we see a big difference between the two protocols. The number of invalidations are far more for the MESI protocol, than the number of bus updates for the Dragon protocol for all benchmarks.

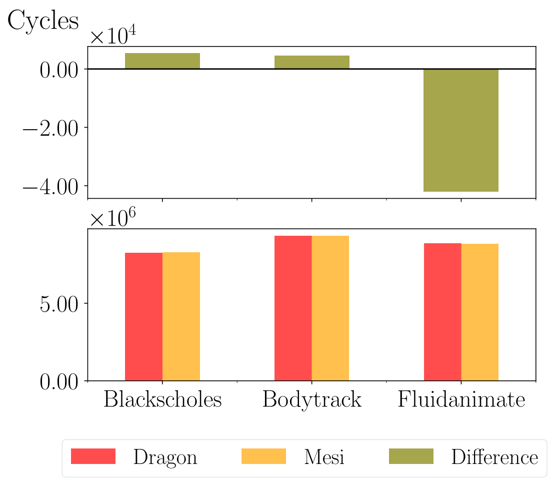


(a) Invalidations or bus updates.

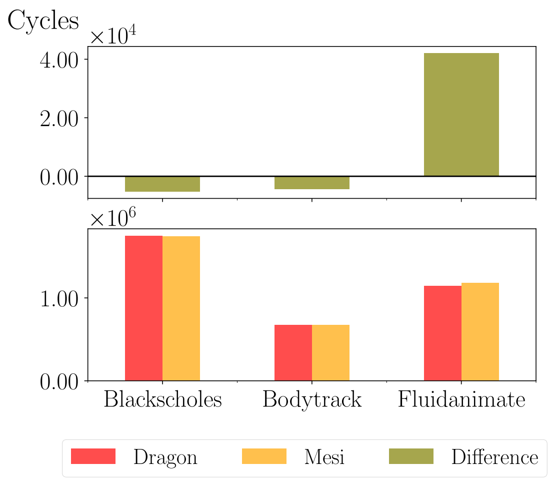


(b) Bus traffic.

Figure 11: The graphs show the number of invalidations or bus updates and the amount of bus traffic



(a) Total of private memory accesses.



(b) Total of shared memory accesses.

Figure 12: The graphs show the number of private and shared memory accesses.

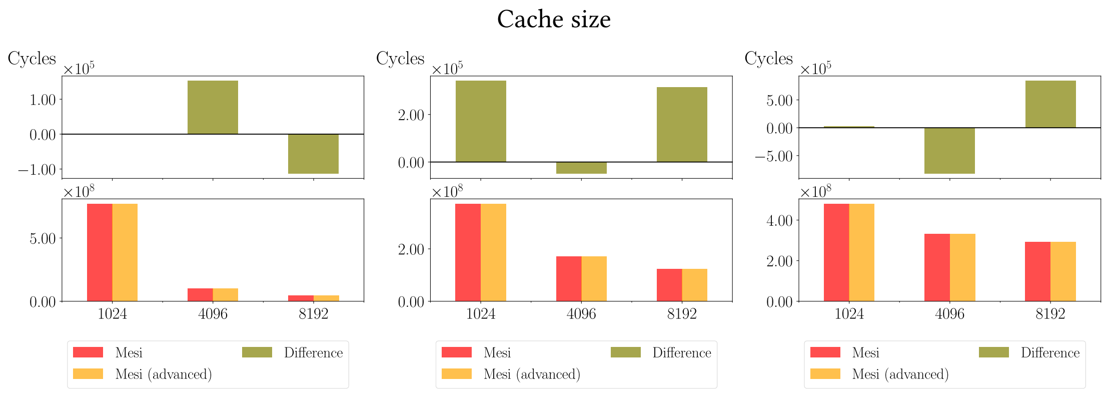
4.2 Advanced Task

The analysis in this section is again based on the default configuration described in the project description and the same tests will be conducted as in the previous section. The difference is that the Dragon protocol is replaced with the advanced version of the MESI protocol. The difference is calculated using

$$P_i^{\text{MESI}} - P_i^{\text{MESI (advanced)}}$$

where $P_i^{\text{<protocol>}}$ is the value for each protocol, for each parameter value i in each plot. A negative difference means that the advanced version of the MESI protocol required more cycles and a positive difference means that MESI was slower than the advanced version.

We see the same overall trend when shifting the cache size, associativity and block size as in the previous section, see Figures 13, 14 and 15. There are more bars in favor of the advanced version of the MESI protocol.

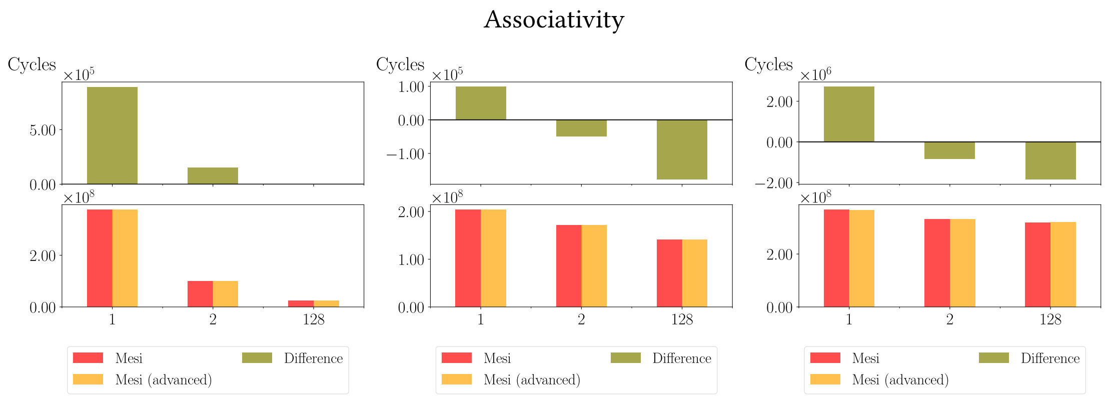


(a) Blackscholes.

(b) Bodytrack.

(c) Fluidanimate.

Figure 13: The graphs show the number of executed cycles when varying the size of the cache. There are three different settings for the cache size, 1024 bytes, 4096 bytes and 8192 bytes. The associativity is 2 and the block size is 32 bytes.



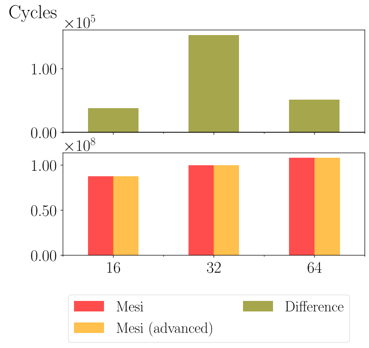
(a) Blackscholes.

(b) Bodytrack.

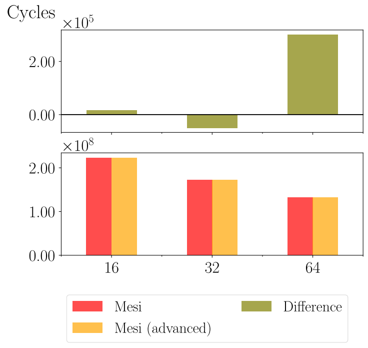
(c) Fluidanimate.

Figure 14: The graphs show the number of executed cycles when varying the associativity. There are three different settings for the associativity, 1 (direct mapped), 2 (2-set-associative) and 128 (fully associative). The block size is 32 bytes and the cache size is 4096 bytes.

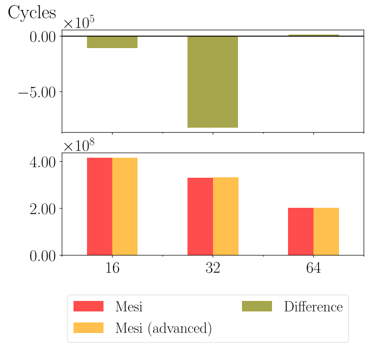
Block size



(a) Blackscholes.

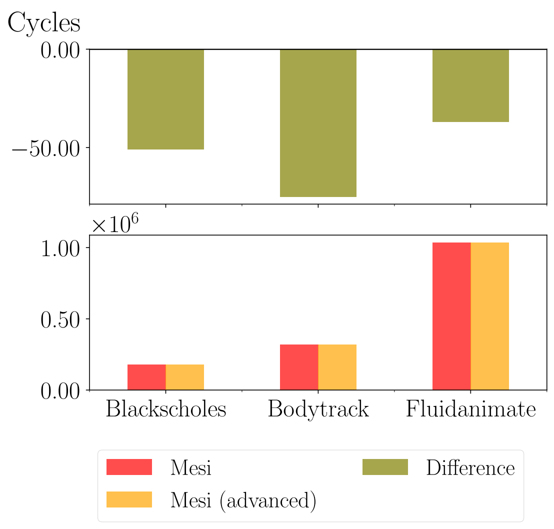


(b) Bodytrack.

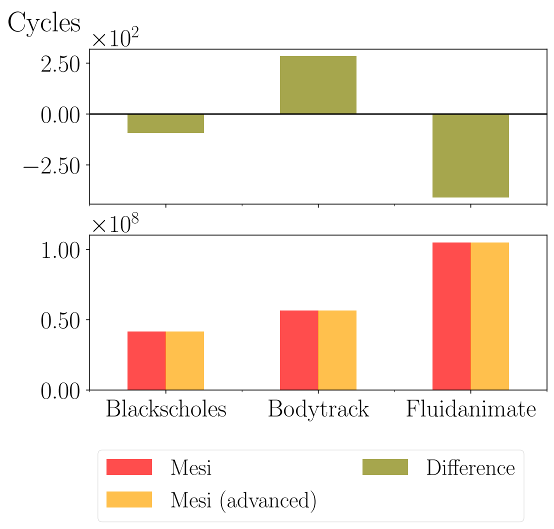


(c) Fluidanimate.

Figure 15: The graphs show the number of executed cycles when varying the block size. There are three different settings for the block size, 16 bytes, 32 bytes and 64 bytes. The associativity is 2 and the cache size is 4096 bytes.

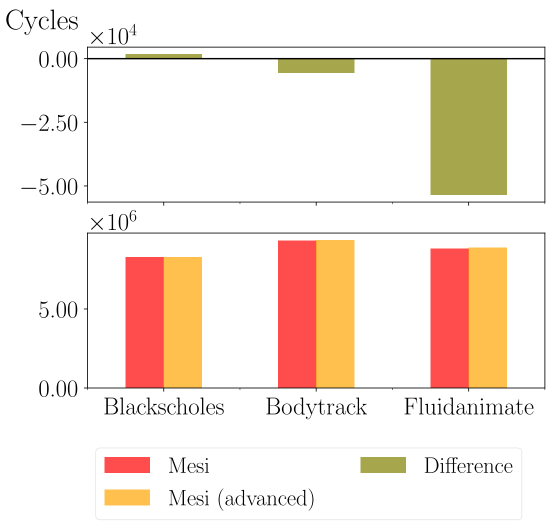


(a) Invalidations or bus updates.

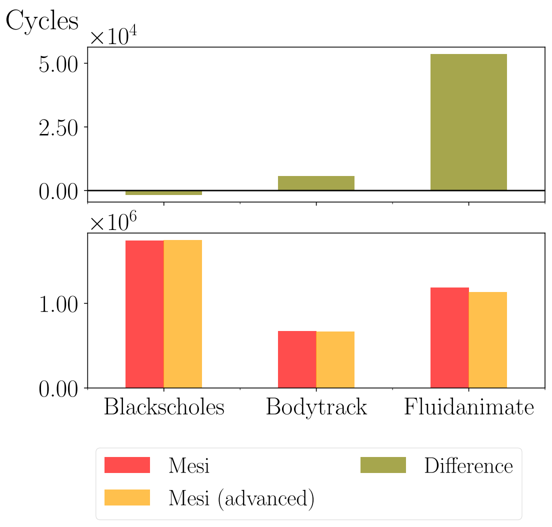


(b) Bus traffic.

Figure 16: The graphs show the number of invalidations or bus updates and the amount of bus traffic



(a) Total of private memory accesses.



(b) Total of shared memory accesses.

Figure 17: The graphs show the number of private and shared memory accesses.

5 Conclusion

6 References

[1] S.J. Eggers and R.H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *The 16th Annual International Symposium on Computer Architecture*, pages 2–15, 1989.

[2] J. R. Goodman and P. J. Woest. The wisconsin multicube: A new large-scale cache-coherent multiprocessor. *SIGARCH Comput. Archit. News*, 16(2):422–431, may 1988.

[3] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1-4):79–119, nov 1988.

[4] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for mimd parallel processors. *SIGARCH Comput. Archit. News*, 12(3):340–347, jan 1984.

[5] Wikipedia contributors. Dragon protocol — Wikipedia, the free encyclopedia, 2022. [Online; accessed 7-November-2022].

[6] Wikipedia contributors. Mesi protocol — Wikipedia, the free encyclopedia, 2022. [Online; accessed 7-November-2022].