Cache Coherence Simulator

Axel Lundberg Lukas Doellerer

Contents

1 Introduction

This report provides the details of how a cache coherence simulator is implemented and shows how three different benchmark traces compare against each other given a certain configuration of the simulator. A cache simulator will try to mimic the behavior of how a real cache would act when presented with a program. However, there is no need to execute a certain program every time the cache simulator is run. Rather, storing the traces of which memory instructions were used in which order would be enough and will reduce the execution time significantly. A cache coherence simulator is not only concerned with simulating the behavior of a cache, but also the interaction between different caches when there is a coherence protocol in place. A coherence protocol is a protocol that states how two different caches in a shared system accesses shared data. Specifically, two caches must never see different values for the same shared data. Two cache coherence protocols will be used in the implementation described in this report: MESI and Dragon. The offer a different take on how to deal with shared data and both has their advantages and disadvantages. The benchmarks are from the PARSEC suite and are the following:

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.

The trace of each benchmark contains has the form shown in Listing 1

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

A label can have the values 0 (load), 1 (store) and 2 (other). The load and store are memory operations and will require the cache. The other instruction is all other kinds of operations. In this simulator, such instructions will be interpreted as a stall in each core for the value specified in the address field. Each benchmark contains traces for four different caches. This, there will be a different sequence of instructions in the form of Listing 1 for every cache. As the simulator is concerned about simulating the behavior of each cache with respect to some cache coherence protocol, actual data is unnecessary. The cache is only concerned what slots the data would have occupied, hence only the address field for each trace.

The implementation should result in 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 are rather long and may take a couple of minutes to run. We have access to a server that can assist us in running the benchmarks. However, while each simulation taking quite some time we will start with running the default configuration for each trace and protocol. Whichever configuration has the best performance will be our baseline where we will try to optimize the parameter settings of the cache size, associativity and the block size.

In addition to measuring the performance of the different traces with respect to MESI and Dragon, an improved version of the Dragon protocol will also be used in the benchmarks. The improved version will optimize a certain aspect of the Dragon protocol with the hopes that the execution time will be faster. The improved version of the Dragon protocol is described in Section 3. This is an advanced task beyond the scope of getting the cache coherence simulator in place.

To be able to measure the performance, the following statistic should be the output of the program

1. Overall execution cycles
2. Distribtion of private data accesses and shared data accesses.
3. For each core
   (a) Number of cycles spent processing other instructions.
   (b) Number of load and store instructions.
   (c) Number of idle cycles, that is, cycles the core has to wait in order for the cache to complete its operations.
   (d) Cache miss rate
4. For the bus
   (a) Amount of data traffic in bytes.
   (b) Number of invalidations or updates

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.

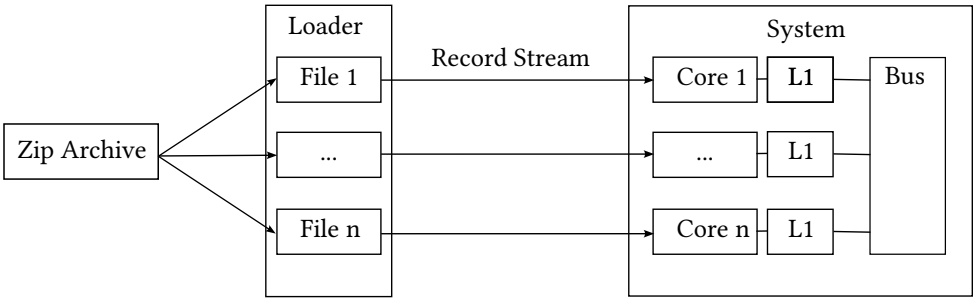


Figure 1: Overview

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 keeps the state of each cache line, the corresponding LRU value for each cache line, which protocol is in use, the scheduled instructions as well as the address layout of the cache. When the cache is initialized in each core the address layout is calculated and consists of the offset length, the index length, the tag length, the set size and the block size. The cache lines and LRU are both represented as a two dimensional vector containing an unsigned integer, where the rows represent the sets and the columns represent the blocks. Note that there is no need to index the words directly as we load the whole block each time we access a word within the block. The scheduled instructions are stored as a deque containing a tuple, the address and the action. There are two valid actions a scheduled instruction can have, read and write. Read operations are inserted at the front of the deque, while the write operations are inserted at the back of the deque. Since the deque is processed from front to back, higher priority will be given to the read operations rather than the write operations.

The cache is updated at every step the core is updated. During an update the cache first checks if there is an active task on the bus that does not belong to the core that is updating. If that is not the case, the cache will return immediately, stalling the cache as the cache has to wait for its last task to finish. Otherwise the cache will pop the front of the deque and try to do either an internal load or internal store depending on the action of the popped instruction. An internal load will search for the given address to see if it is already present in the cache, a hit, and perform the neccesary steps to update the state for the cache line depending on which protocol which is in use. If there is not cache hit, the cache will check if a writeback is needed given the current value on the cache line. An internal store will also search for the given address to see if it present in the cache, but will in the case of a cache miss push a read action to the deque to fetch the current address before writing to it — write-allocate policy. Both the internal

load and internal store will put the needed action on the bus if the bus is not occupied and it is required by the state transition.

2.1.5 Bus

The bus keeps track of the current task on the bus. There can only be one such task at one time. A task has the following form:

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

Thus, the task contains the id of the core that 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. That is, a bus action like `BusUpdShared` is only valid for the Dragon protocol, so when the MESI protocol is used, this action is 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),
}
```

Each bus action state contains an address and the number of bytes used for the bus exchange. Each update cycle the bus proceeds to advance the bus transaction if there is one.

2.1.6 Protocol

The protocol is implemented as a trait, where a trait defines shared behavior. This is very similar to how interfaces work in other languages than Rust, with some minor differences. For example, traits cannot have fields. The shared behavior for protocol trait is defined like:

```
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 protocol is stored like

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

and operations are for example invoked like

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

This is really useful as the cache does not need to know which protocol is used, but relying on the fact that the underlying protocol has some defined behavior for the invoked operation. In every update cycle, as described earlier, there are three phases

1. **Step**. The step phase correspond to either a `read()` or `write()`.
2. **Snoop**. The snoop phase correspond to `snoop()`.
3. **After snoop**. The after snoop phase correspond to `after_snoop()`.

2.2 Protocol

Like described in section 2.1.6, each protocol needs to implement a set of method to satisfy the 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 processor initiated transitions for the MESI protocol can be found in Figure 2. However, this is a bit different from the transition diagram found on Wikipedia [6]. 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 all other transistions will be described in this section.

TODO: describe cache hit for I.

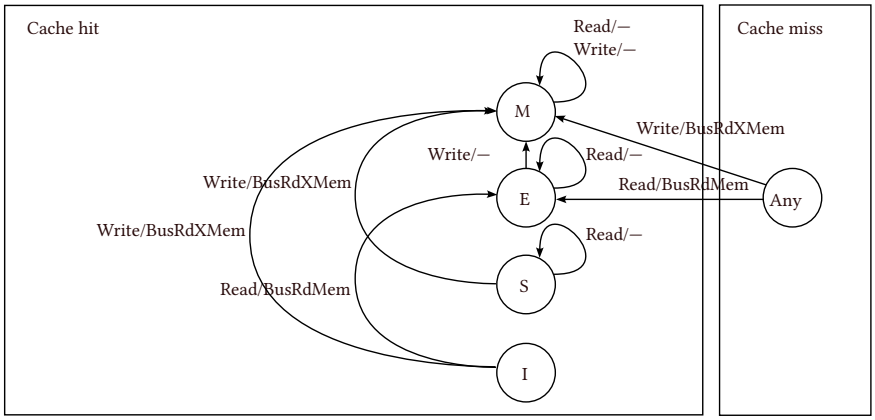


Figure 2: MESI. The step phase of the update cycle.

$I \rightarrow S$. 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 occured. Another core that is currently holding the same cache line will snoop the bus transaction 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 transistion from E to S in Figure 4.

$I \rightarrow E$. When a cache miss occurs, the core will transistion 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.

$I \rightarrow M$. 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.

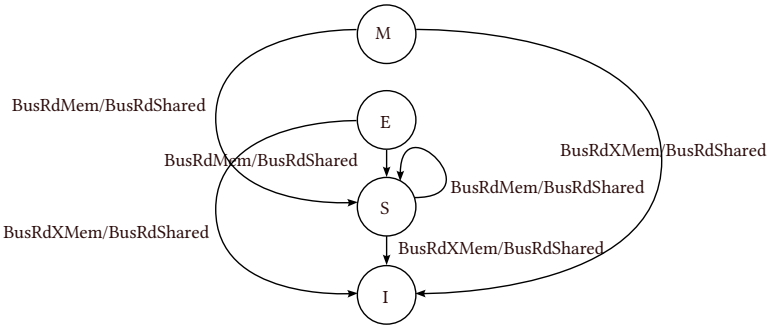


Figure 3: MESI. The snoop phase of the update cycle.

$S \rightarrow S$. When a read hit occurs, no state transition will occur and no bus transation will be issued. Thus, the state will stay in S.

$S \rightarrow M$. 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.

$E \rightarrow E$. When a cache hit occurs while a core is reading to a cache line that is in E, the cache line will remain in E and no bus transaction will be issued.

$E \rightarrow M$. 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.

$M \rightarrow M$. When a cache hit occurs while a core is writing or reading to a cache line that is in M, the cache line will remain in M and no bus transaction will be issued since it can be sure that no other core has a copy of the cache line.

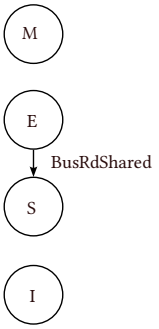


Figure 4: MESI. The after snoop phase of the update cycle.

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 happend 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 ($I \rightarrow S$). 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 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 noticable difference is that there are no transitions from an "invalid" state to state Sc, Sm and M. Like the previous section all transitions will be described in this section. Even if "invalid" is not a state in the Dragon protocol, "Any" will denote a cache line that is not currently in E, Sc, Sm or M. Thus, the cache line is guaranteed to miss in this "state".

TODO: cache-to-cache

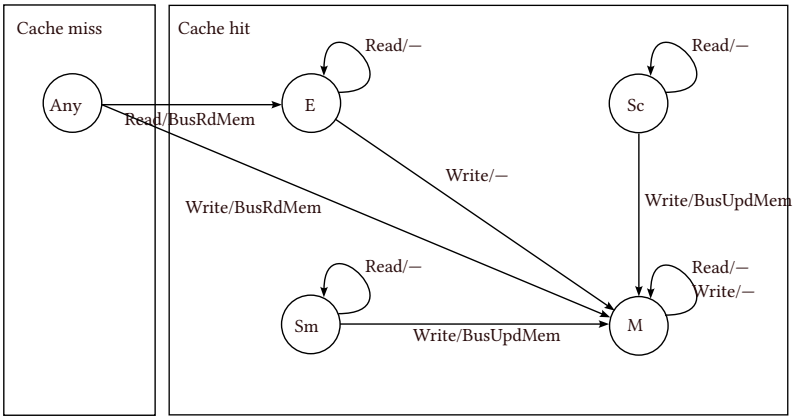


Figure 5: Dragon. The step phase of the update cycle.

Any → *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 change their states accordingly.

Any → *M*. When a cache miss occurs while a core is writing to a cache line the cache line will enter state M. 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 M.

Any → *Sm*. When a cache miss occurs while a core is writing to a cache line the cache line will enter state M. A 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, so these cores will during the snoop phase notice the bus transaction. and change their states to Sc. Any cache line that was in Sm or M will have to propagate the data. This is represented as a penalty where all cores are stalled as no data is transmitted.

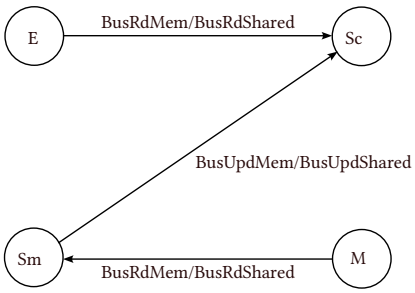


Figure 6: Dragon. The snoop phase of the update cycle.

$E \rightarrow M$

$Sc \rightarrow M$

$Sm \rightarrow M$

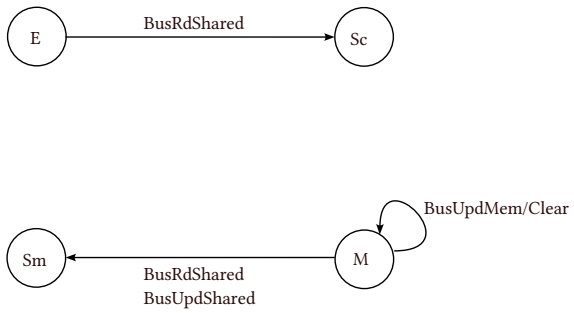


Figure 7: Dragon. The after snoop phase of the update cycle.

$M \rightarrow M$. When a cache hit occurs while a core is writing or reading to a cache line in state M, the cache line will remain in state M and no bus transaction will be issued.

$E \rightarrow E$. When a cache hit occurs while a core is reading to a cache line in state E, the cache line will remain in state E and no bus transaction will be issued.

$Sc \rightarrow Sc$. When a cache hit occurs while a core is reading to a cache line in state Sc, the cache line will remain in state Sc and no bus transaction will be issued.

$Sm \rightarrow Sm$. When a cache hit occurs while a core is reading to a cache line in state Sm, the cache line will remain in state Sm and no bus transaction will be issued.

2.3 Testing

To verify that all state transitions work, a set of unit tests for the MESI protocol and the Dragon protocol have been constructed. The tests are manually constructed to check if a certain transition happend given a cache line's current state and whether another core is occupying the same cache line. These tests are all successful. However, the tests are not comprehensive in that they include all possible scenarios. 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. Only one core is used in these tests. Thus, this should in principle verify that the cache implementation is correct, not neccesarily that the cache coherence implementation is correct.

TODO: make sure the tests are working

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 be 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 need to 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 quantiative 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

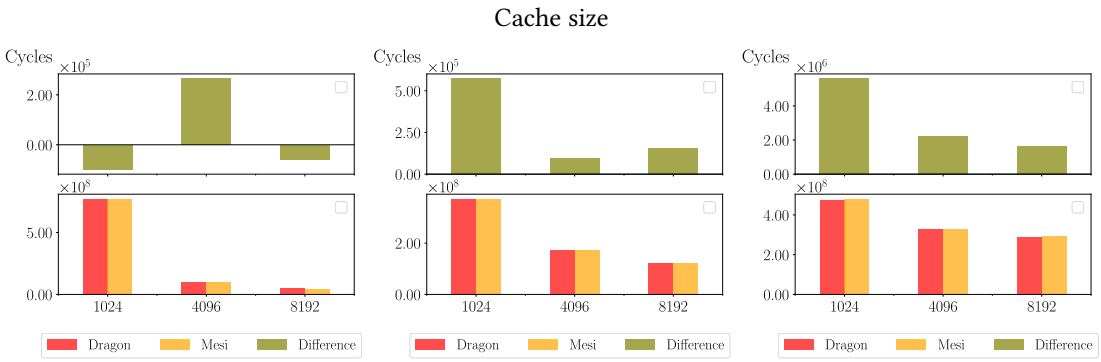
The analysis in this section is based on the default configuration described in the introduction. The MESI protocol and the Dragon protocol will be evaluated on every benchmark trace while varying one of either the cache size, the block size or the associativity. Each plot contains two subplots, one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

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

where $P_i^{\text{<protocol>}}$ is the value for each protocol, for each parameter value i in each plot. 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,

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

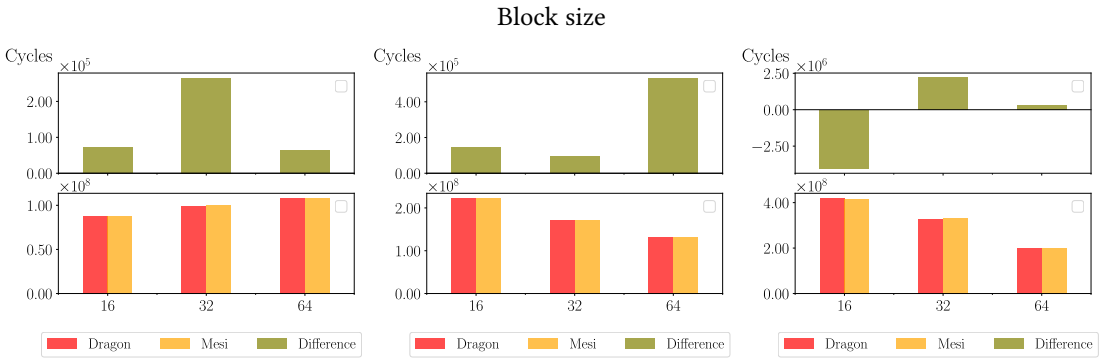
without accounting for the advanced tests. Figure 8 shows the performance for each protocol when varying the cache size.



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

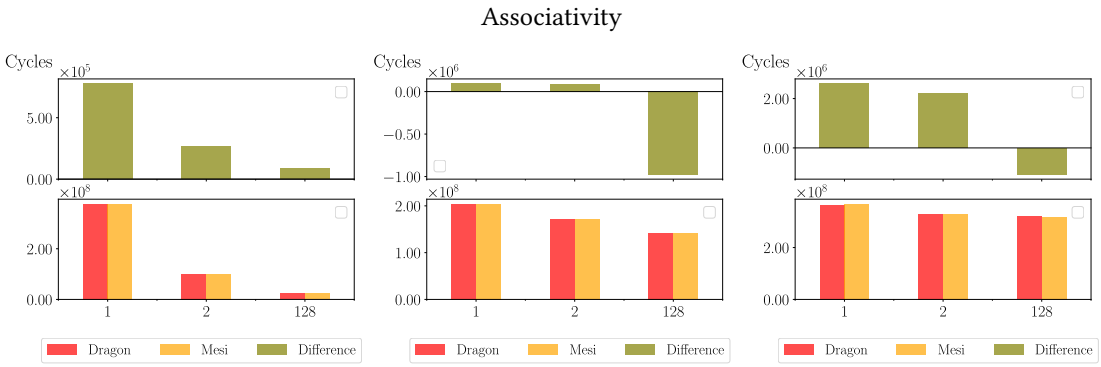
Figure 8: The graphs show the number of executed cycles when varying the size of the cache. There are three different settings for the cache size, 1024 bytes, 4096 bytes and 8192 bytes. The associativity is 2 and the block size is 32 bytes.

We can see that ...



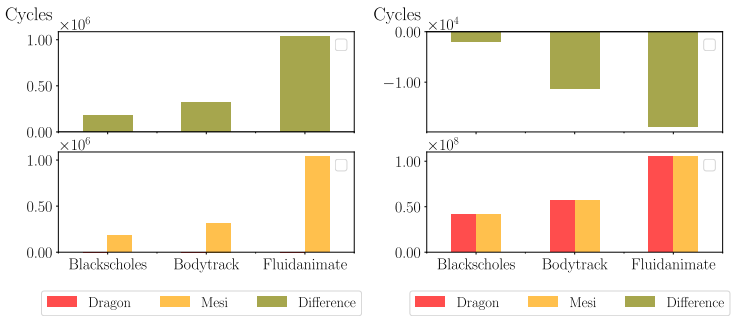
(a) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

Figure 9: 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) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

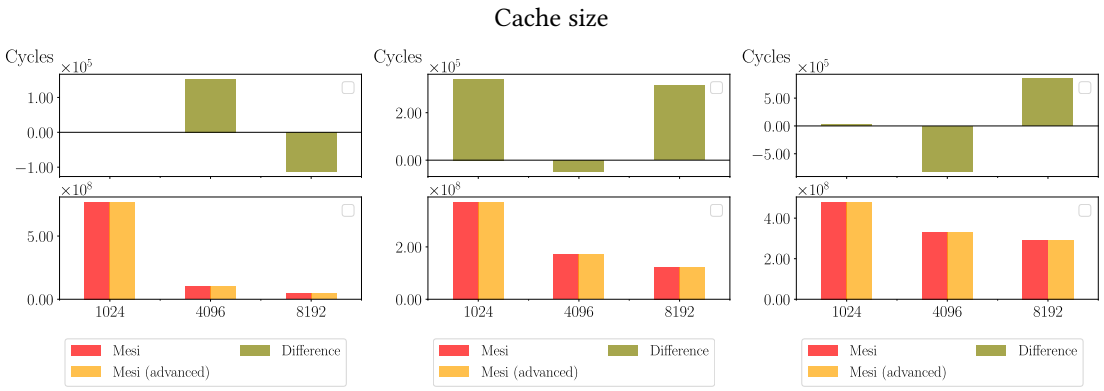
Figure 10: The graphs show the number of executed cycles when varying the associativity. There are three different settings for the associativity, 1 (direct mapped), 2 (2-set-associative) and 128 (fully associative). The block size is 32 bytes and the cache size is 4096 bytes.



(a) Invalidations. (b) Bus traffic. (c)

Figure 11: The graphs show the number of invalidations, bus traffic and ... for all the protocols and benchmark traces with the default settings.

4.2 Advanced Task

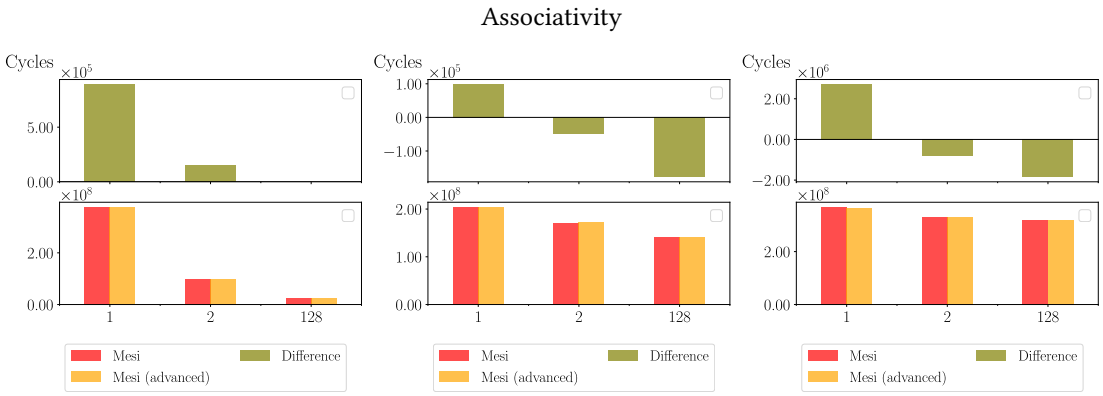


(a) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

Figure 12: 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.

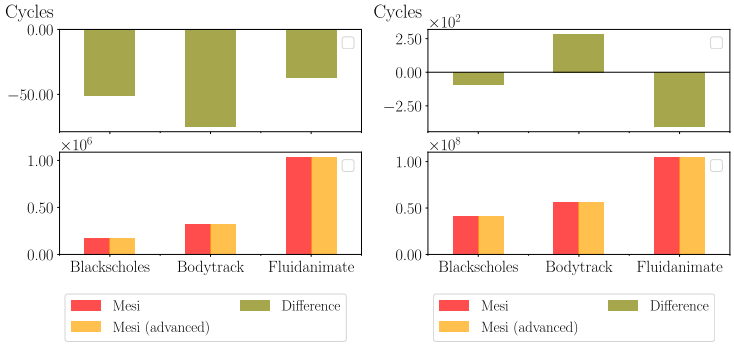
(a) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

Figure 13: 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) 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.



(a) Invalidations. (b) Bus traffic. (c)

Figure 15: The graphs show the number of invalidations, bus traffic and ... for all the protocols and benchmark traces with the default settings.

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].