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 behaviour 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 behaviour 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 Blach-Scholes Partial Differnetial Equation.
2. **bodytrack**. Body tracking of a person.
3. **fluidanimate**. Fluid dynamics for animation purposes with 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 behaviour 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> 4048 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

There are a couple of assumptions that need to be made about the behaivour of the simulator. There are a couple of assumptions clearly stated in the project description and we have further expanded the list of assumptions along the way. The core assumptions stated in the projcet description are the following:

1. Memory address is 32-bit.
2. The word size is 4 bytes.
3. A memory reference accesses 32-bit (1 word) of data.
4. Only the data cache will be modeled.
5. Each processor has its own L1 data cache.
6. L1 data cache uses write-back, write-allocate policy and LRU replacement policy.
7. L1 data caches are kept coherent using cache coherence protocol.
8. Initially all the caches are empty.
9. The bus uses first come first serve arbitration policy when multiple processor attempt 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. 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.

Further, we have to make some additional assumptions

1. Instruction scheduling happens instantly => the clk cycle that a "other" is scheduled is already the first cycle in which it is reduced.
2. Writing always takes one cycle to hit the cache.
   (a) Write Hit => 1 cycle delay
   (b) Write Miss => 1 + (cache_miss_penalty) delay
3. A bus update always only transmits a single word, bus flushes always transmit a whole block. Flushes always also go to main memory and therefore require at least 100 cycles. Updates can go to other caches only, making them faster with about 2 cycles.
4. Dragon protocol: Bus flushes are only required for write backs. As long as the copy stays in the cache, every "flush" in the diagram is replaced with a shared update action. No data is written to main memory.
5. Cache write-allocate: every write checks if the address is currently in cache. If not, it schedules a load and restarts the check afterwards. Only if the check is successful (hit), the write is attempted. If another cache invalidates the cache line between the read and the write, then the read has to be repeated.
6. Bus wait cycles are only counted beginning in the clock cycle AFTER the task was put on the bus. This means that a writeback requires in total 101 cycles until the next action can be performed: 1 cycle to schedule the writeback and 100 cycles for the bus to finish the writeback to memory.
7. Caches block during their own bus transactions => the cache waits until its bus transaction is finished until it commences with further steps (this is required to restart the transaction in case something fails)
8. Other caches may listen to the bus during flushes to main memory and can therefore directly update / read their new value. This means that a bus read that causes a flush (MESI) only takes the time that is required to flush to main memory (which is > than shared read time).
9. Our MESI is Illinois MESI.
10. Dragon Protocol: Replacement of Sc blocks is not broadcast
11. Dragon Protocol: 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 existing caches holds the requested tag.
12. Dragon Protocol: (From Sm, Sc) On a PrWr, a BusUpd is scheduled. 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.
13. If a valid version of the cache line is in the cache (hit), only single words are updated / read / write. If there is not valid version in the cache (miss) or invalid, the full cache line (block_size) is transmitted via the bus. For write-backs, also the full cache line is transmitted.
14. A cache hit occurs if the cache contains a block that contains the requested address. It is not affected by the protocol state of the line. This means that accesses to invalidated cache lines are also counted as cache hits.

1.2 Methodology

The cache coherence simulator is implemented using Rust. Rust is a compiled language with similar performance to that of C and C++. The Rust compiler is very strict and induces a certain coding style

that once it compiles, one often can be sure of that unexpected behaviour will not happend. While it often is harder to get a prototype working, hopefully less time will be spent on debugging rare edge cases.

Alongside with the cache coherence simulator, unit tests and integration tests will be conducted to ensure the correct behaviour of the simulator.

2 Method

The following section describes how the different components work as well as a more detailed view of how the MESI and Dragon protocols work.

2.1 Components

Figure 1 shows an overview of how the different components in the simulation are structured. We can see that 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 cache each and is connnected to the bus.

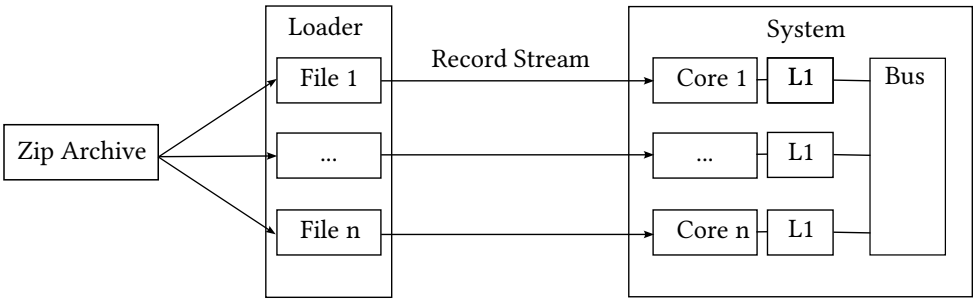


Figure 1: Overview

2.1.1 Record

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

```
0 0x817ae8
```

a record will be created in the following form

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

2.1.2 System

The system keeps the state of all the cores and make sure that each core is updated. The system will create as many cores as needed to satisfy each file unpacked from the zip archive. The update step for each core consists of three stages: `step`, `snoop` and `after_snoop`. The `step` stage parses the new instruction into a record and updates the state given the record. The `snoop` stage will snoop the bus. If the bus currently have an active task and the task's issuer is not the core that is snooping the core may update its state given the active task and current state for that cache line. The `after_snoop` stage is a cleanup stage done after the snooping stage. It consists of changing the state of a cache line depending on the outcome of the snoop stage, that is, if it turns out that the cache line should be shared.

2.1.3 Core

The core keeps the state of the cache and all the records.

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 behaviour. 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 behaviour 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 behaviour for the invoked operation. The core

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 can be found in Figure 2. However, this is a bit different from the transition diagram found on Wikipedia. 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 the other transistion will be described in the following section. The transition from invalid (I) to shared (S) is instead modeled using the step phase (Figure 2) and the

after snoop phase (Figure **??**). 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. In the same cycle

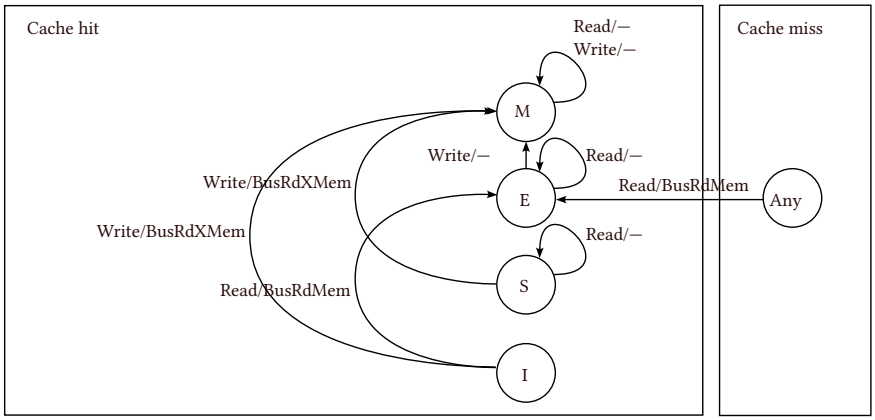


Figure 2: Mesi. The step phase of the update cycle.

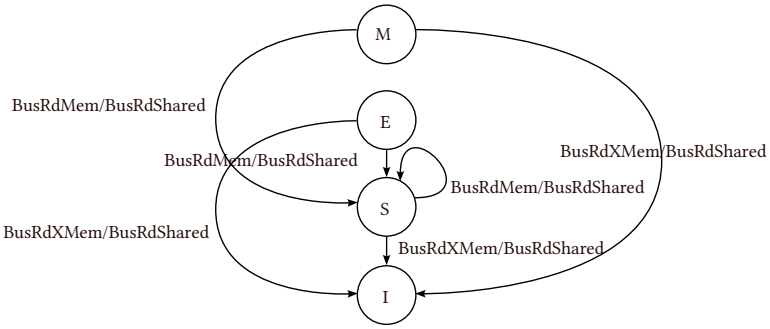


Figure 3: Mesi. The snoop phase of the update cycle.

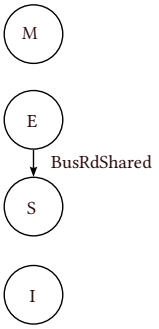


Figure 4: Mesi. The after snoop phase of the update cycle.

2.2.2 Dragon

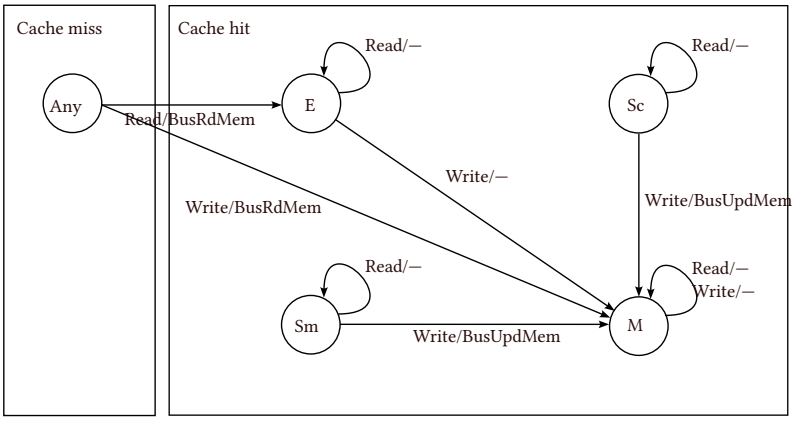


Figure 5: Dragon. The step phase of the update cycle.

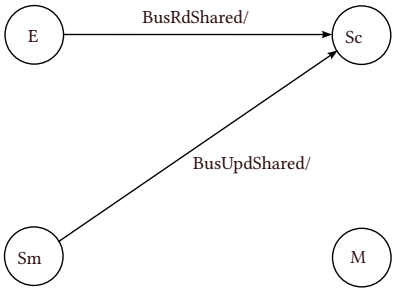


Figure 6: Dragon. The snoop phase of the update cycle.

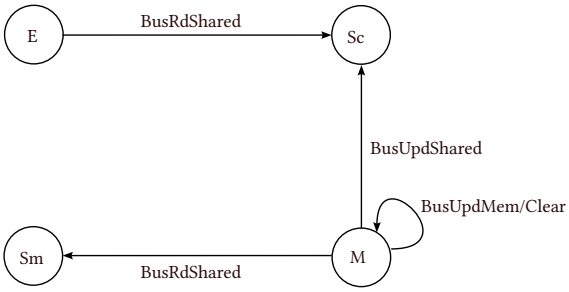


Figure 7: Dragon. The after snoop phase of the update cycle.

2.3 Testing

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.

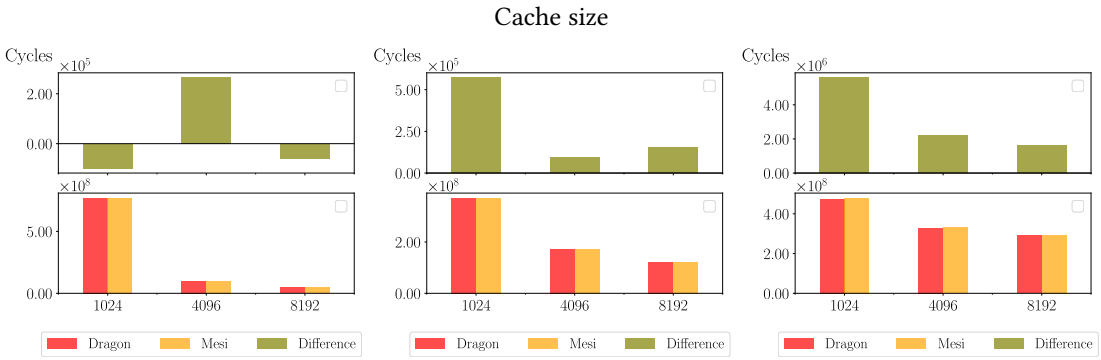
Suppose we have a system with multiple cores that each hold the same block in their caches. We also assume that at some point one of the cores initiates a write to this block and therefore sends an invalidation signal to all the other caches. If one of the other cores now were to issue another read to 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.

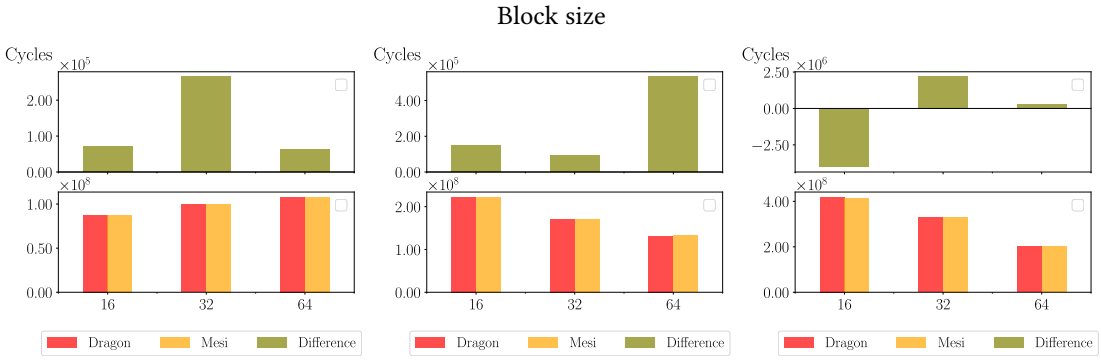
4 Results

4.1 Quantitative Analysis



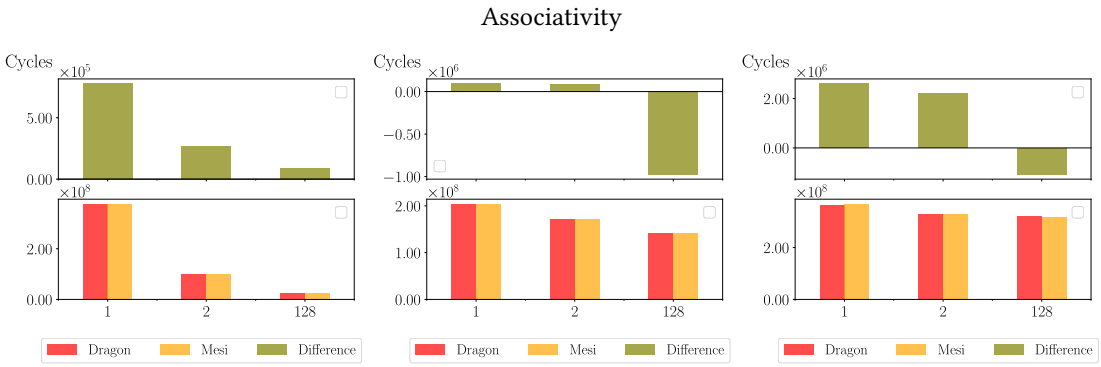
(a) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

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.



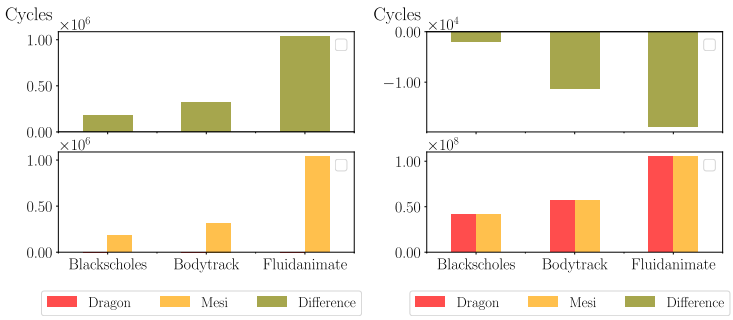
(a) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

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.

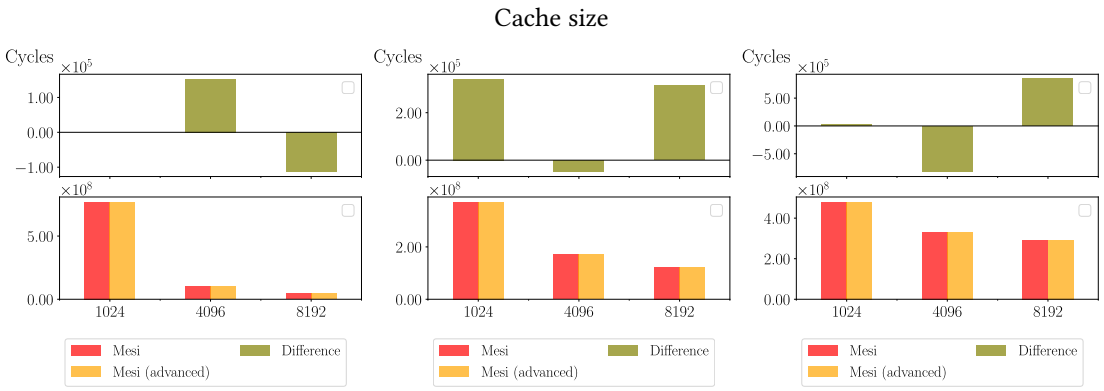
Figure 10: The graphs show the number of executed cycles when varying the associativity. There are three different settings for the associativity, 1 (direct mapped), 2 (2-set-associative) and 128 (fully associative). The block size is 32 bytes and the cache size is 4096 bytes.



(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

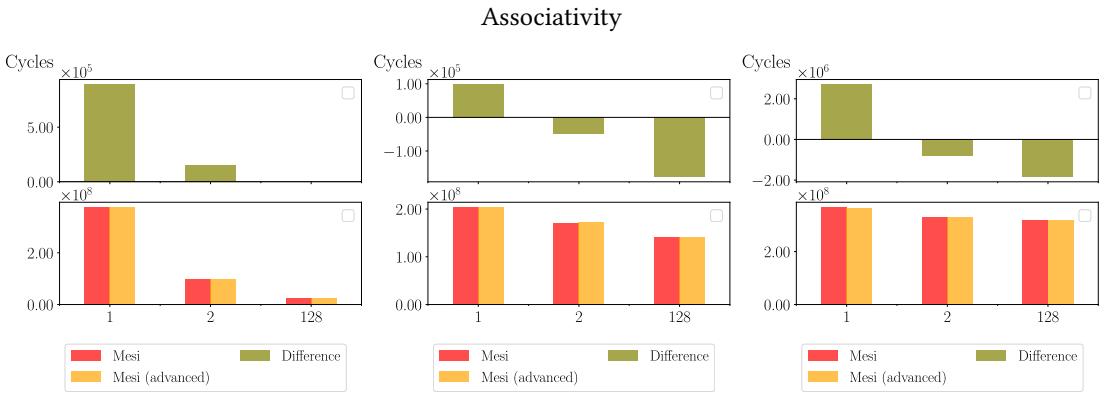


(a) Blackscholes. (b) Bodytrack. (c) Fluidanimate.

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.

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.