Cache Coherence Simulator

Axel Lundberg Lukas Doellerer

Contents

1 Introduction

1.1 Assumptions

TODO: Make cleaner

1. Memory address is 32-bit. Note that the address shown in the example trace file is 24 bits because the 8 most significant bits are sometimes 0. So the address 0x817b08 is actually 0x00817b08
2. Each memory reference accesses 32-bit (4-bytes) of data. That is word size is 4- bytes.
3. We are only interested in the data cache and will not model the instruction cache.
4. Each processor has its own L1 data cache.
5. L1 data cache uses write-back, write-allocate policy and LRU replacement policy.
6. L1 data caches are kept coherent using cache coherence protocol.
7. Initially all the caches are empty.
8. The bus uses first come first serve arbitration policy when multiple processor attempt bus transactions simultaneously. Ties are broken arbitrarily.
9. The L1 data caches are backed up by main memory — there is no L2 data cache.
10. 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. PrWriteMiss does not exist at the protocol level. Every write miss first allocates using a read. Therefore, every PrWriteMiss is translated to PrRdMiss -> Write
4. 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.
5. 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.
6. 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.
7. A write to a cache line does not update its LRU-counter

8. 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.
9. 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)
10. 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).
11. Our MESI is Illinois MESI.
12. Dragon Protocol: Replacement of Sc blocks is not broadcast
13. 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.
14. 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.
15. 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.

1.2 Rust

2 Method

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.

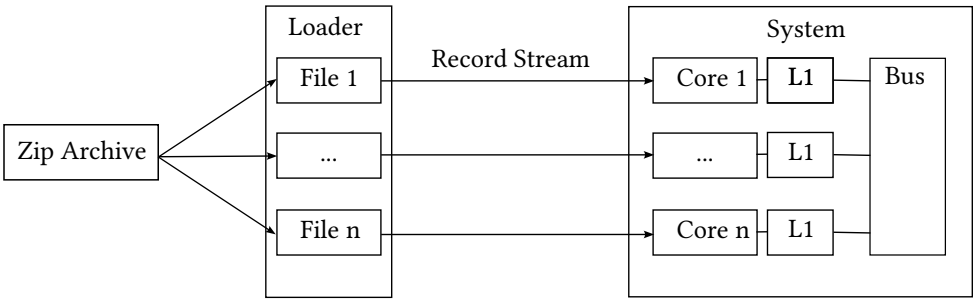


Figure 1: Overview

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.

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

2.2.2 Dragon

2.3 Advanced Task

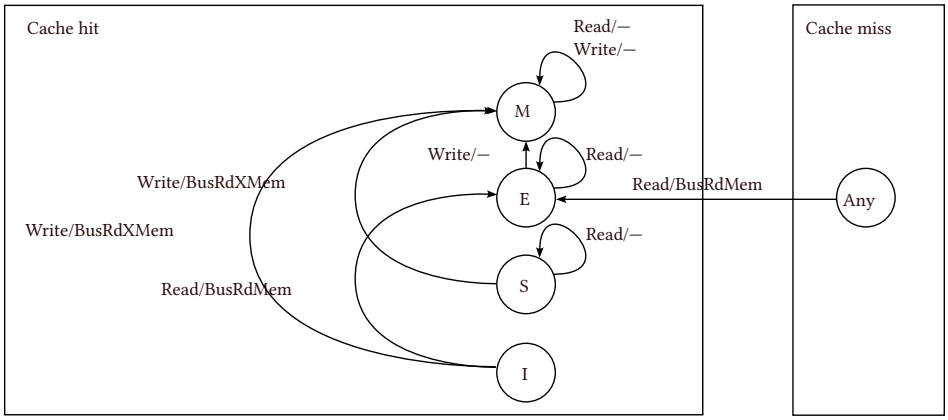


Figure 2: Mesi

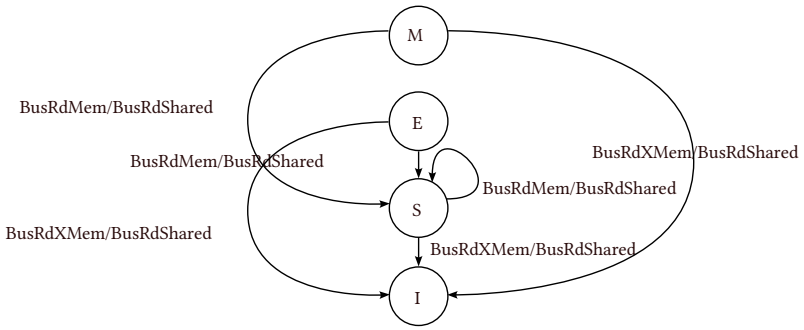


Figure 3: Mesi snoop

3 Results

3.1 Quantitative Analysis

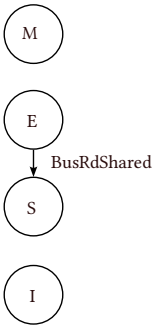
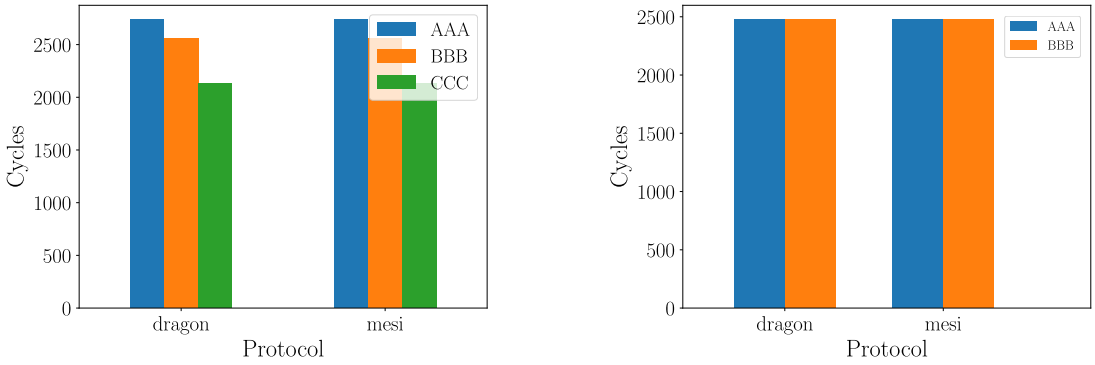


Figure 4: Mesi after snoop



(a) Associativity

(b) Cache size

Figure 5: Placeholder graphs

4 Conclusion