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

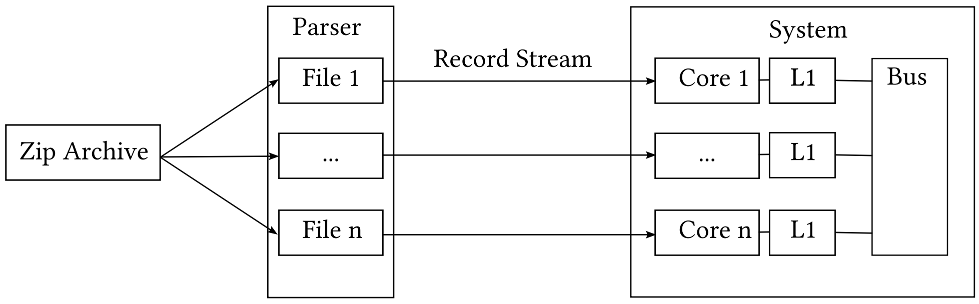

Figure 1: Overview

2.1 Components

2.1.1 System

2.1.2 Core

2.1.3 Cache

2.1.4 Bus

2.2 Protocol

2.2.1 MESI

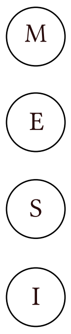

Figure 2: mesi

2.2.2 Dragon

2.3 Advanced Task

3 Results

3.1 Quantitative Analysis

4 Conclusion