Cache Organization in C++

by Anshik Sahu and Sanya Mittal

May 11, 2023

§1 Introduction

This report describes the implementation of a simulator for an n-way set associative cache in C++. The simulator has two levels of caching, L1 and L2, and is designed to read trace files and assign requests to the L1 cache. The L1 cache then sends read/write requests to the L2 cache, which interacts with DRAM. Both the L1 and L2 caches keep track of their own counters, such as the number of reads, writes, hits, and misses. At the end of the simulation, for a given trace file, the program prints out the statistics for both caches on the console, including the number of reads, misses, writes, miss rates, and the number of writebacks from each cache. We aim to study the miss rates of caches over large data sets.

§2 Implementation

We have designed a structure to represent the cache and used a Hierarchy structure to represent the memory hierarchy. We have abstracted the concept of main memory and not explicitly designed it in our implementation. This is an implementation for a single-core processor, following a write-allocate policy, which means each time a block is evicted, the block is first fetched from the main memory and loaded into the cache, and then the write operation is performed on the cache block.

§2.1 Cache Organization

In an n-way set associative cache, the memory address is typically divided into three fields: the tag field, the index field, and the byte offset field. The index field is used to locate the set in the cache where the requested block might be present. In each set there are n ways in an n-way associative cache. The tag field identifies the unique memory block that is being accessed and is used to check whether the requested block is present in the cache or not. The byte offset field identifies the byte within the cache block where the requested data is located.

§2.2 Dirty Block

A cache block is marked as dirty when it has been modified by a write request and memory has not been updated. A dirty block is different from one that's not in this way - if a block is marked dirty and it has to be evicted then it first has to be written back to the next level of memory, if that level is the main memory then we can successfully evict the block, else we must keep that block marked as dirty.

§2.3 Validity

Each cache block is associated with a validity bit that indicates whether the block contains valid data or not. When a block is fetched from the main memory into the cache, its validity bit is set to ”valid” to indicate that the block contains valid data. Any subsequent read or write operation on that block can be serviced from the cache, avoiding the need to fetch the block from the main memory again. In the case of multi-processors, the valid bit is also changed when some other processor tries, but in this case, the invalidity of a bit represents a garbage value that has not been cleared after some previous execution.

§2.4 Read Requests

Whenever a read request is encountered, it is first sent to the L1 cache, block address is computed according to its memory address. If the block is present in the L1 cache, read request is successful and a read hit occurs. If not, the read request is sent to the next level of the hierarchy, that is L2, and if the block is found there then it is brought back to L1 and then the read request execution is successful. When the block is not found in L2 as well, then it is fetched from memory, and brought back to L2 as well as L1.

§2.5 Write Requests

When a write request is encountered, the block address is first computed according to its memory address and sent to the L1 cache. There would be three cases -

1. If the block is present in the L1 cache and is not marked as ”dirty” then the write request will be successfully executed and data would be written to the memory block.

2. If the block is present in the L1 cache and is marked as ”dirty”, meaning it has been modified since it was brought into the cache, then the modified data is written back to the next level of the hierarchy, which is the L2 cache. Once the data has been written back to the L2 cache, the write request is then executed successfully in the L1 cache, and the block is marked as ”dirty” in the L2 cache. Whenever it will be written back to the memory the dirty label would be changed.

3. If the block is not present in the L1 cache, the request would be passed on to the L2 cache, if the block is found there if it’s not marked as dirty the block would be brought back to L1 and write request would be executed, else it would first be written back to memory and then the block would be brought to L1 cache and write request would be successful.

After each write, the block in that particular cache would be marked as dirty, and would be updated only when the data is written back to the next level of memory.

§2.6 LRU Policy

The least Recently Used Replacement policy is used for block eviction which means that the block which was least recently used would be evicted. We have implemented this by maintaining a parameter lru with each cache block, which would be incremented each time the block is used, and during eviction the block with the least lru would be evicted. The LRU of all empty blocks are initially set to 1 so that they will be considered first for replacement in the priority order.

§2.7 Inclusivity

We have also implemented the cache with inclusivity and without inclusivity. For maintaining inclusivity we have evicted a block from L1 if it is evicted from L2. This ensures that L1, at all times is a subset of L2. In terms of performance for these trace files, we did not observe any significant change with our design decisions. Code with and without inclusivity have been submitted in separate files.

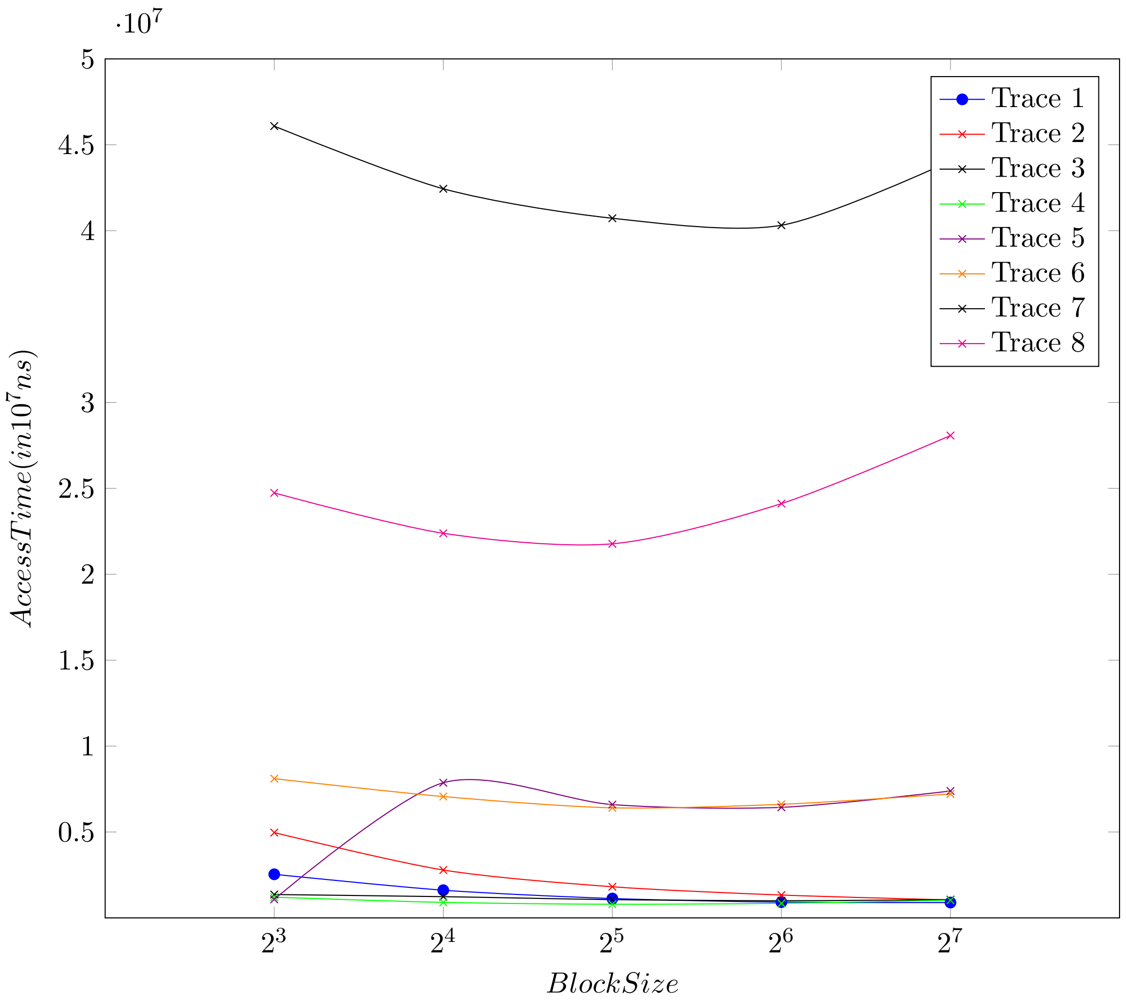
§2.8 Access Time Computation

In access time computation we have taken the miss penalty to be 0 ns. Miss penalty is the extra time required for a failed access.

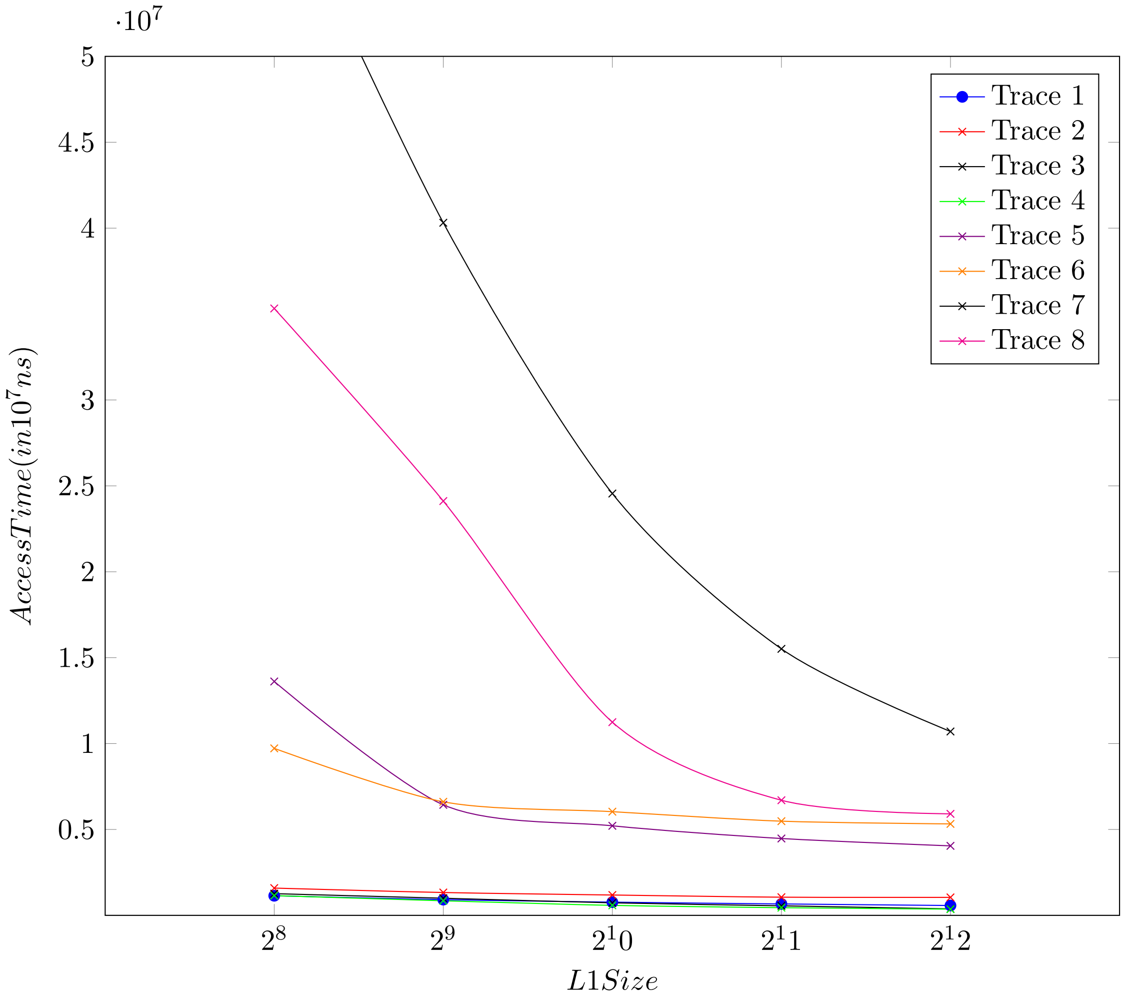
§3 Comparison of miss rates over varying parameters

§3.1 Varying Block Size

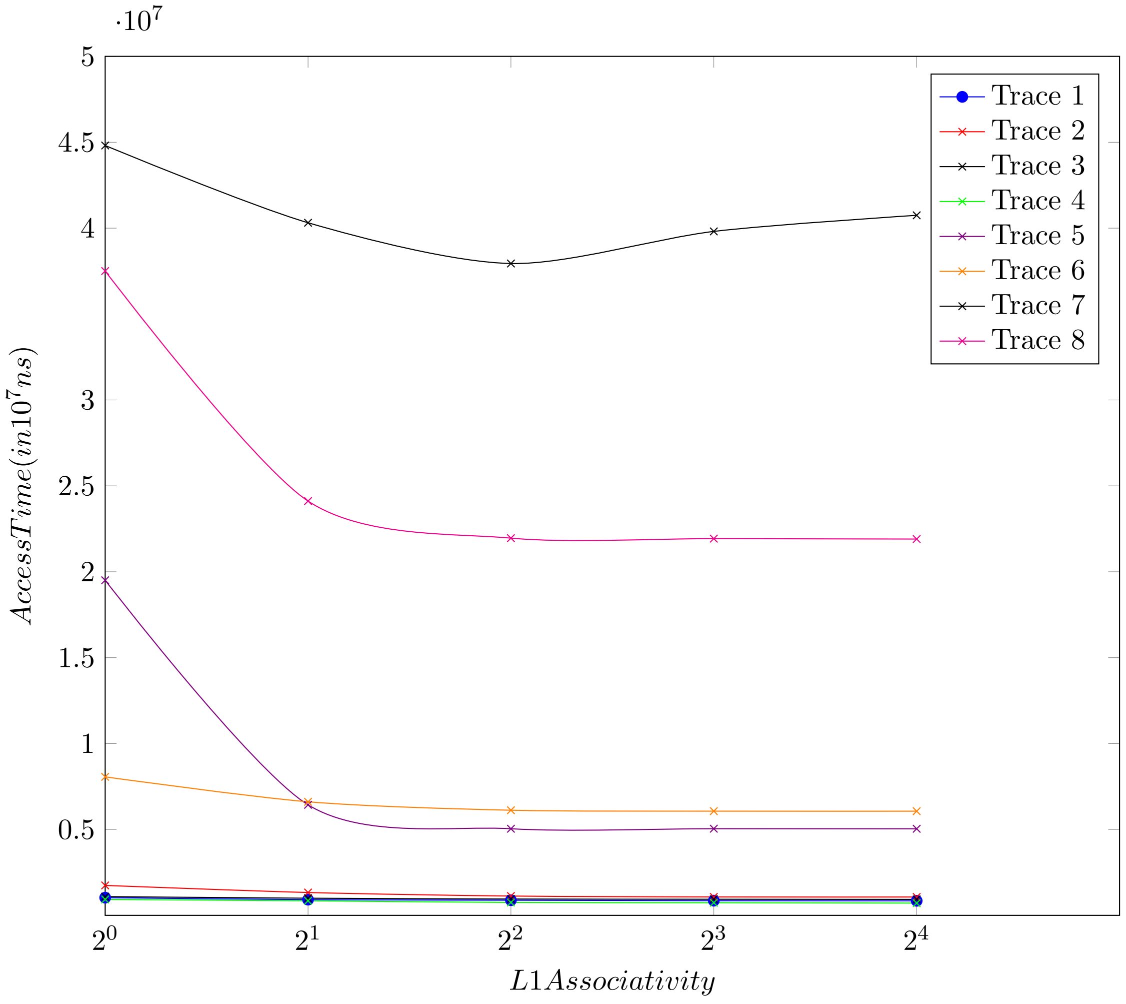
Keeping L1 Size,L1 Associativity, L2 Size and L2 associativity constant set to their default values i.e. 1024, 2, 65536, 8
Block size is varied between 8, 16, 32, 64, 128 and the following data is observed.



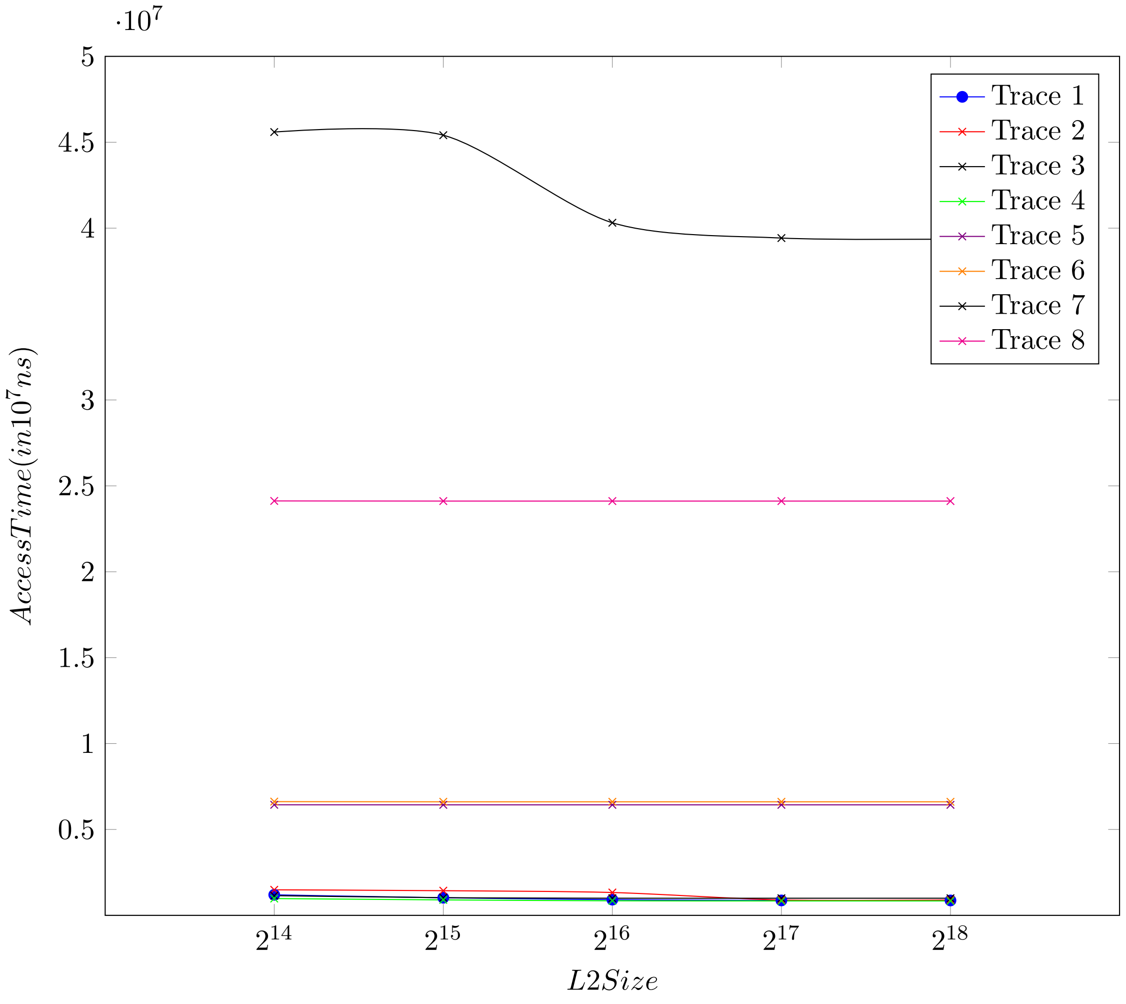
§3.2 Varying L1 Size



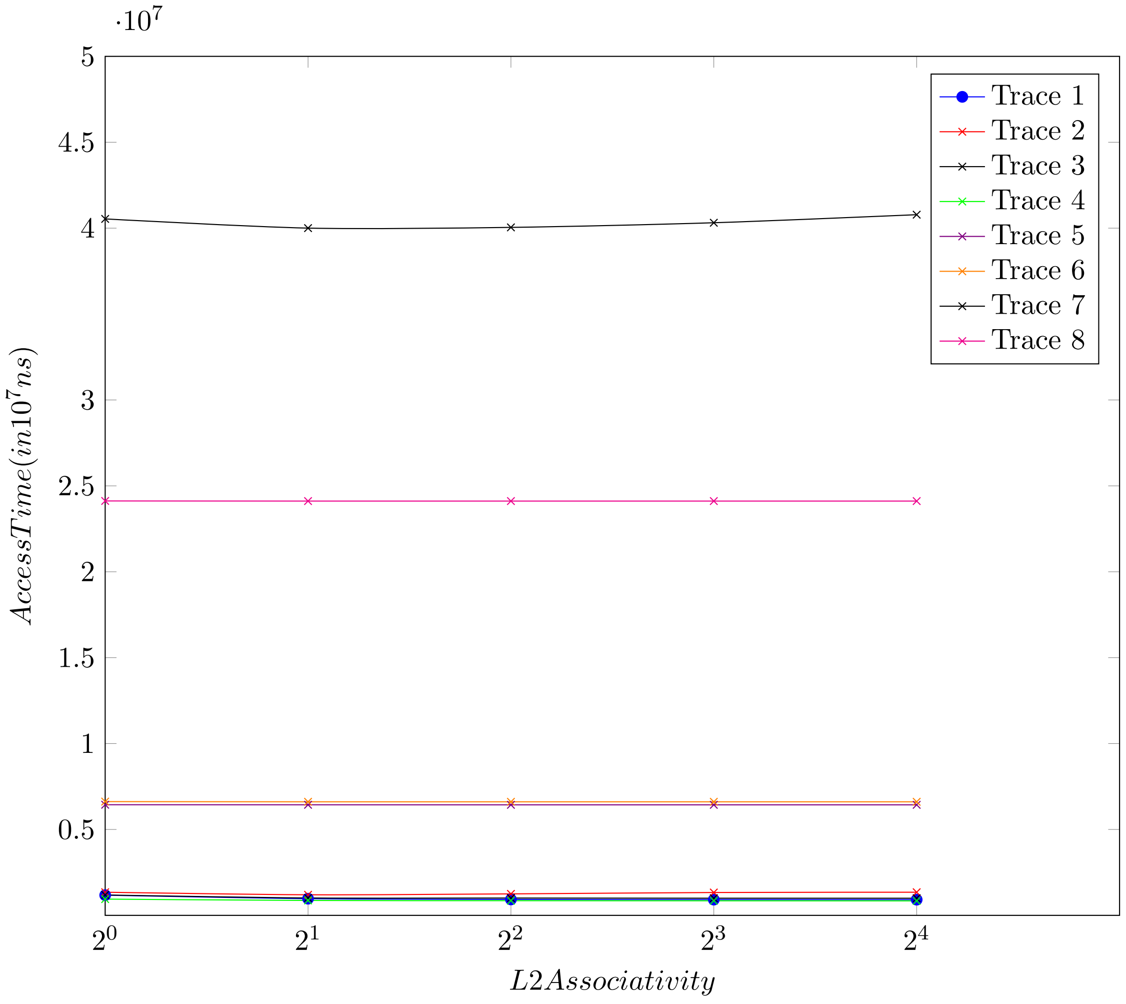
§3.3 Varying L1 Associativity



§3.4 Varying L2 Size



§3.5 Varying L2 Associativity



Observations : 1. Due to temporal locality, when we increase the cache size the access time decreases.

2. When we increase the number of sets in the L1 cache while keeping all other parameters same, the overall time falls.

3. The aforementioned statement is not true for L2 cache because we are enlarging the cache by increasing the number of sets, and increasing sets reduces capacity misses, but in this instance, there are very few capacity misses, leading to a very little decrease in overall time.

4. As we raise the Block size while maintaining the cache size, the total time first drops as the spatial locality rises, then rises again because the decrease in the number of sets outweighs the effect of spatial locality. This can be used to point out that a direct-mapped and fully-associative cache do not have the minimum access time and instead, it is better to have a n way cache where n is decided by cache size

5. As we improve the Associativity of the cache, the overall time lowers.

6. The effects of change in L1 parameters are more dramatic as compared to that in L2 as L1 is closer to the processor, this can be generalized to deduce that increasing

DRAM size has little to no effect on execution time.

7. The access time is almost constant with the change in L2 associativity as the cache is sufficiently large so that the number of sets is not a bottleneck and the increase in number of ways has almost no effect as processor can only access a subset of L2 through L1(except when there is only 1 way, the time is more as the associativity of L1 is more).

8. Increase in L1 size show large deduction in access time as this increases the directly accessible data to the processor.

9. There is an increase in the access time of Trace 5 with increase in block size. This is due to addresses being mapped to the same index which increases with decrease in number of sets.

10. The above point also explains the sudden decrease in access time of Trace 7 with increase in L2 size.

11. Trace 7 has the highest access time in all cases which can be due to higher number of instructions and less spatial locality.

Design Decisions :

1. We have abstracted DRAM and not explicity structured it. 2. We have designed the cache for a single processor.

3. We have used write allocate policy.

4. We have used LRU Policy for eviction of blocks.

5. We have assumed that the miss penalty is 0 in the computation of Total access time.

§4 Conclusion :

The work split in this assignment has been:
Sanya Mittal : 2021CS10565 : 50%
Anshik Sahu : 2021CS10577 : 50%