Cache Simulator

COL216 – A3 - Part A&B

Vedant Sameer Talegaonkar
2022CS11603

Introduction:

In this assignment, the aim is to build a cache simulator which given a memory trace, will mimic the behavior of a cache with a specified configuration and output the resulting statistics.
I have written this code in C++ with its Standard Template Library (STL).

Parameters to judge effectiveness:

The parameters that I have used to analyse the cache effectiveness are:
1. **Total cache size:** Ideally, we would like to minimise this to reduce the memory costs, however, a very small cache leads to minimal hits and therefore poor performance.
2. **Miss rate (load and store):** A cache is meant to be a bridge between processor and disk memory, so a lower miss rate would mean that the main memory is not accessed often, leading to better performance.
3. **Clock cycles per given trace:** This is a straightforward way of measuring performance of a cache. Clearly, lesser clock cycles leads to better performance.

There are many more parameters which decide the effectiveness in real life – cost of comparators to check tag, hardware convenience and so on. However, these parameters have not been quantified in our simulation, therefore I decided to not use them to analyse cache configurations.

Factors affecting these parameters:

1. **Associativity:** Depending on the number of positions that a block can occupy in the cache, the chances of getting a hit are determined.
2. **Block Size:** Owing to the principle of spatial locality, if a word is requested, its neighbouring words in memory are likely to be requested as well. Therefore, this factor has a direct impact on number of hits.
3. **Write-through/Write-back:** Depending on the logic, this choice has an impact on main memory accesses.
4. **Write-allocate:** If a program uses a given word multiple times, having it allocated in the cache is a huge benefit – however, it also increases miss penalties.

I have not included replacement policy in my analysis, owing to its complicated behavior.

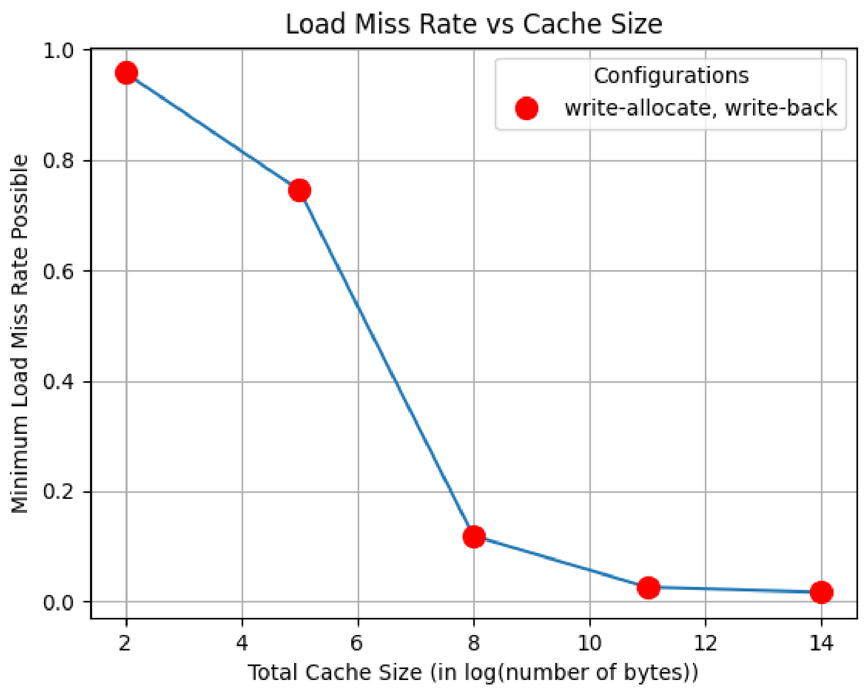
Analysis:

Depending on these given conditions, I decided to apply the following tests:
1. For any given **total cache size**, I found three configurations (number of sets, blocks/set, bytes/block, write-allocate, write-back/through), among which first will give the **minimum load miss rate**, second gives the **minimum store miss rate**, and third gives the **minimum** clock cycles for a given trace file.
2. Then, I applied this testing method on cache sizes ranging from 4 bytes to 16 kB, hence giving us information over various sizes, and drew the plots for each parameter vs cache size.

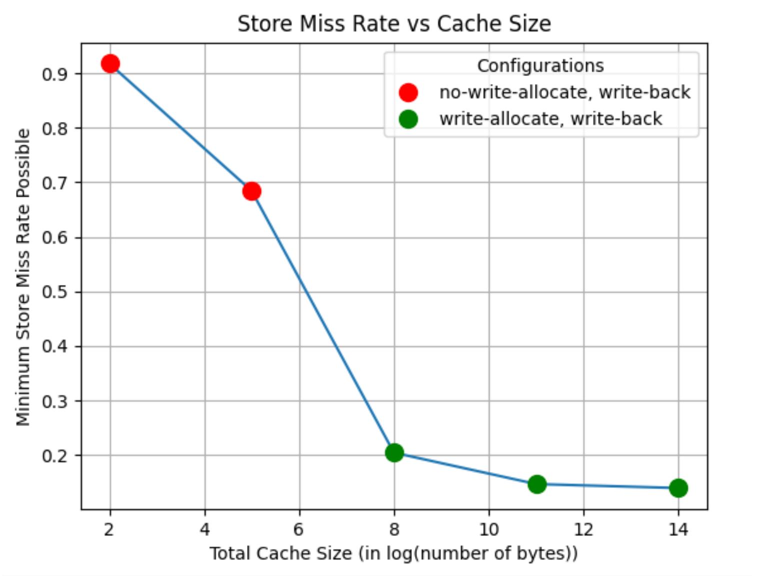
For all of these values, the trace file is *gcc.trace*, which contains more than 500 thousand loads/stores, leading to intensive testing.

Results:

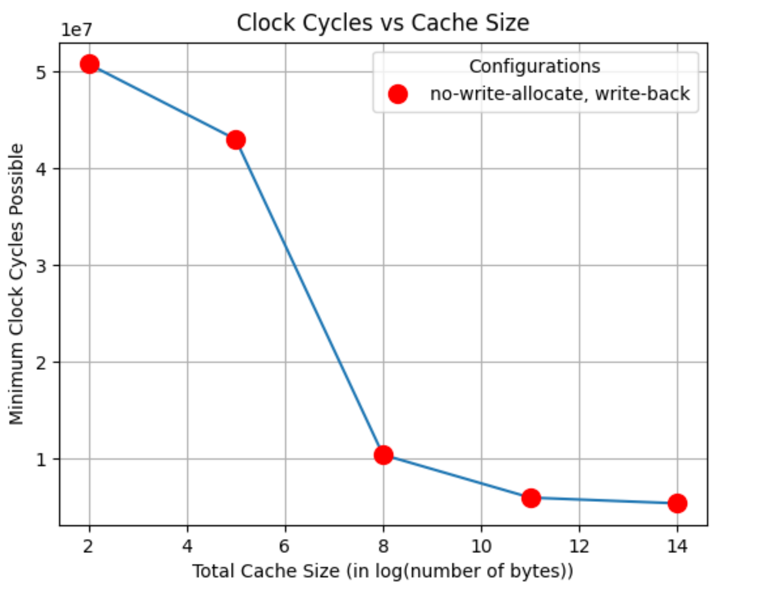
In the following results, all of these points in the graphs are configurations which are fully-associative. This is because in general, fully associative caches lead to lower miss rates and main memory accesses over direct-mapped caches.



Above is the graph of **load miss rate** vs **cache size**. We can observe the decreasing trend of load miss rate with cache size, with a very good ~1% miss rate for cache of size 16 kB.
However, to achieve a balance with decent load miss rate and a moderately sized cache, we can see that 256 bytes cache with write-allocate and write-back configuration is the best one so far.



In this graph of **store miss rate** vs **cache size**, we can observe a similar trend to the previous graph. Therefore, the cache which gets us the most bang for our buck is again, the 256 bytes cache with write-allocate and write-back.



Here, we observe a different scenario – the 256 bytes cache is favoured again, however, the configuration with minimum clock cycles is no-write-allocate, write-back.

Conclusions:

Based on our testing method, we can conclude that there is no one single best cache which can work for all situations.
- For memory traces with a skewed proportion of loads/stores, we conclude that fully associative 256 bytes cache with write-allocate and write-back is the best cache.
- For memory traces with a uniform distribution of loads/stores, we conclude that fully associative 256 bytes cache with no-write-allocate and write-back is the best cache.