INF-2200: Computer architecture and organization

Assignment 3 - Cache Simulator

Tristan Natvig & Tarje Carlsen

November 21, 2025

1 Introduction

In this report it is discussed how the cache simulator is the most optimal design for a quicksort benchmark dataset used in a previous assignment. The cache is specifically configurated to run at optimal efficiency while still remaining a low hardware cost. This is done with carefully adjusting the different aspects of the cache until it reaches a level that both satisfies the performance needed while still maintaining a "as low cost" as possible.
The benchmark used for the cache simulation is take from another assignment and is a sorting algorithm using quicksort. The quicksort code sorts a dataset of size 1000 integers and iterates a total of 50 000 times. This creates a logfile of roughly 17mb which fits well for a small cache simulation without giving us to much fluctuation when running random replacement policy.

The memory access pattern for the quicksort benchmark is a mix of sequential, reverse and recursive access behaviours. Where as the left and right scans both uses spatial locality as the memory accessed are located close together in memory. Read hits and misses dominate this phase, as elements are compared against a pivot value, which all are read from memory. Swapping in quicksort happens when elements on the wrong side of the pivot are found. This involves writes to two array positions and can therefore lead to write hits and misses. The arrays is recursively divided into smaller subarrays where the access becomes less predictable, and leads to a decrease in temporal locality and an increase in cache misses at deeper level of the recursion.

2 Method

2.1 Implementation & design

The cache simulator models a two level memory hierarchy consisting of: a read-only level 1 instruction cache L1I, a read/write level 1 data cache L1D and a unified read/write level 2 cache L2. The simulator follows a design where each cache is implemented as a configurable component. The simulator was written in C and integrated with the provided CPU file.
Each part of the caches are built using structs. There are structs for each line, each set, counts for hits and misses, and a struct for handling the tag,index and offset of adresses. This is done to make the cache easily configurable as each different part has its own section. Using a struct for the splitting of adresses makes it possible to split the adresses with a function and return the struct which then can be used for different functionalities. Finally there is a struct for caches that take in all the necessary elements to build the cache. Memory accesses are driven by binary tracefiles, produced using valgrind and traceconverter. The simulator routes each instruction, fetch, data read and write requrests through the memory hierarchy based on the access type. The cache is designed to easily be editable with new configurations for new tests. All of theese options are defined at the top of the code and contain easily changeable values. All the different types of settings, like write policy, associativity, replacement policy are defined as enums at the top where they use their names for what configuration to use. For example if the cache should run on a write back policy then the write policy would be set to WRITE_BACK.

2.2 Calculating hit-rate

To track the performance of the cache simulator, we implemented a mechanism for counting read and write hits and misses for each cache level. This was done by integrating a dedicated Hit_Miss struct within each

cache instance, which maintains separate counters for:

- Read hits
- Read miss
- Write hits
- Write miss

Since the L1 instruction cache L1I is read only by design, only read hits and misses are tracked for this level. Write operations are never routed to L1I and are therefore ignored in its statistics.
The main function responsible for counting hits and misses is CacheLookup. This function checks whether a given address exists in the cache with a valid tag. If the address is found, it is counted as a hit; otherwise, it is a miss. Based on the operation type instruction fetch, data read, or data write, the appropriate cache level and counter are updated.
Each memory operation is processed in a hierarchical order, starting from the top-level cache L1I or L1D, and only propagates to the lower level L2 on a miss. This allows us to collect detailed statistics for each layer independently.
At the end of the simulation, all statistics are printed to the terminal in the memory_finish function. This includes both raw counts and calculated hit rates in percentage, making it easy to analyze the effect of different cache configurations during testing.
The formula used for calculating the percentages for hit rates are as follows: Hit rate = (number of hits / total accesses) * 100. This calculation was done to make optimization of the cache as easy to optimize as possible.

2.3 Calibrating the cache

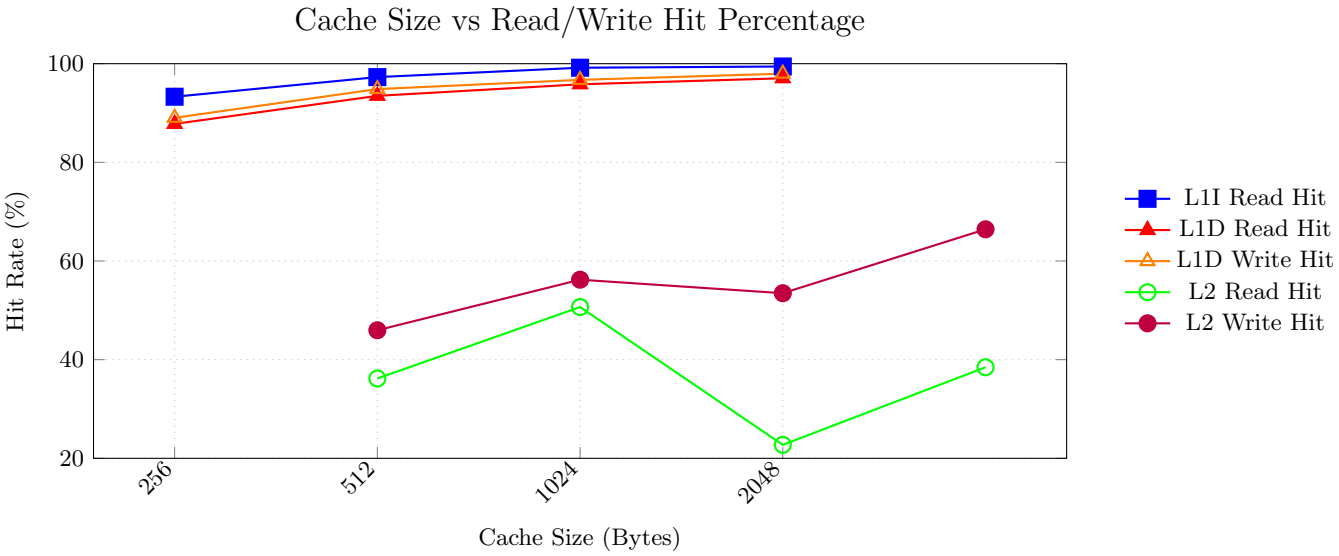
To ensure that the cache simulator produced correct results, we created a controlled trace file named test.tr. This file contains a small set of 10 carefully selected memory instructions designed to trigger known cache behaviors, such as reads, writes, hits, and misses across different cache levels.
Each instruction in the trace was analyzed manually to determine the expected outcome based on the current cache configuration. This included calculating tag, index, and offset values, then comparing them with the initial state of the cache. The expected results were written down and compared with the output from the simulator.
The purpose of this calibration phase was to verify the internal logic of the simulator before using it on large benchmark datasets. The results from running the simulator with test.tr aligned perfectly with our expectations. These expected results are documented in the fasit.txt file, which serves as a reference for correctness.
This calibration step proved valuable for debugging early in the development, as it allowed us to isolate and test the cache logic without the noise of larger benchmarks. To reproduce this test, the simulator can be run with the command: cachesim test.tr. The output will show the number of hits and misses per cache level, making it easy to verify correctness.

2.4 Testing

The parameters that was changed during the testing was the size of the caches, the assosiativity, the mapping design, line size and write policy. This how the cache was evaluated:

1. The parameters was set to a realistic starting point. A complex state with a large cache and line size and a high assosiativity.
2. The size and complexity of the cache was decreased step-wise until the hit-rates where undesireble.
3. The testing started by running the cache and writing down the parameters and the resulting hit rates.
4. For every run the size and complexity was increased and the results was saved.
5. When the increase in hit rate stalled the testing was complete.

3 Results



The graph shows how hit rate increases when the cache size increases. It also show that the hit-rate of the L2 cache is a bit more fluctuating. Take into account that the L2 cache size is always double of that of the L1 caches. This means that the 256b L1 caches are run with the 512b L2 ect. There is an appendix in the repo with all the test results. The graph is only from the caches with set associative mapping, a line size of 64b, associtivity of 2 and a Write Back write-policy.

4 Discussion

When looking at the test results we have done tests way beyond what we thought to be necessary. For a cache to be optimal it needs to balance a good hit rate with a low costing implementation. The factors affecting the cost that are explored in these tests are the total cache size, line size, mapping design, assosiativity and write policy.
When choosing the most optimal cache design we set up a few requirements with focus on cost efficiency and realism.
Knowning that set associative mapping is the mapping design that provides the best balance between cost and data integrity this was chosen as a must for the chosen cache. Write back is the most cost effective write policy and was also a requirement.
Bus width was not implemented in our cache and therefore was not changed in the testing.
A line size of 64 is the most commonly used and thereby the most realistic[1].
To keep it realistic the size of the L2 cache was double that of the L1 cahces.[2]
With the above mentioned requirements the most optimal cache design was chosen as the least complex cache that had above 90% both read and write hit rate.

In our tests this was the cahce with the following paramterers:

Cache Level	Size (bytes)	Associativity	Mapping	Replacement	Line Size	Write Policy
L1I	512	2	Set Associative	Random	64	Write Back
L1D	512	2	Set Associative	Random	64	Write Back
L2	1024	2	Set Associative	Random	64	Write Back

Table 1: Cache Configuration

Which produced the following results:

Cache Level	Read Hit (%)	Write Hit (%)
L1I	98.4	N/A
L1D	93.91	95.07
L2	37.66	57.66

Table 2: Cache Hit Rates for Read and Write Operations

It was also prevalent that when increase the size and assosiativity any more only gave miniscule increases to the hit rate while significantly increasing the overall cost.

The reasons behind the stalling can be many. When looking at the design and the benchmark log the workload of the tracefile might be small enough that 512 and 1024 bytes is sufficent. It would of course be difficult to increase the hitrate any more than above 99% and any increases would come at a cost outweighing the benefits.

When starting working on the cache the plan was to implement a 4-way set assosiative cache. During testing there was a miniscule difference between the 2-way and 4-way set, and the biggest difference was a higher miss-rate in the L2 cache. It was therefore concluded that a 2-way design would be sufficient, if not preferable when evaluating the cost of a 4-way design. Further testing could render the implementation of the set assosiative mapping unnecessary for this specific logfile but we made the assumption that a set assosiative mapping would and should outperform direct mapping.

The replacement policy was chosen as random because of its simple and cheap implementation. When evicting randomly the need for logic that keeps track of memory accesses is gone. The logfile does not seem to perform badly in our cache which could indicate that it does not have a pattern that necessarily favors LRU, but further testing is needed to conclude anything about the choice of this policy.

When observing the graph it is clear that with the spesific dataset used in the testing there is a clear connection between cache size and hit-rate. It is also easy to explain the slow incline of the L2 hit-rate; the more hits in the L1 caches, the fewer requests are sent down to the L2 cache. Most of the relevant data fits in the L1 caches.

The logfile from our benchmark has hundreds of thousands of addresses and determening why our benchmark is optimal would be time consuming to investigate. The reason that this uncomplicated cache works well can be because the logfile includes many repeating and adjecent addresses which gives high spatial and temporal locality. This means that a smaller size can still fit the most relevant and often used addresses. With a lower locality the cache would need to be bigger and have more ways to keep the hit rate equivalent.

5 Conclusion

The configuration from Table 2 seems to be the most optimal cache design for the specific dataset used in this project. It has a good hit-rate while still being fairly simple to implement. The hardware cost is small and the cache has a relativly low access time. The data structure is cheap, uncomplicated and small in size and while still being realistic in a real hardware scenario. This is as mentioned specific for our dataset and may not be applicable in other cases.

References

[1] CPU Cache and Cache Lines: The Hidden Keys to High-Performance Code - Medium 13. November, 2025
`https://medium.com/@umeshcapg/cpu-cache-and-cache-lines-the-hidden-keys-to-high-performance-code-f567024822e3`

[2] CPU Cache Basics - Larapulse Technology 13. November, 2025
`https://dev.to/larapulse/cpu-cache-basics-57ej#:~:text=L1%20Cache:%20Typically%20ranges%20from%2016KB%20to,to%2032MB%20or%20more%20in%20high-end%20processors.`

[3] Learning Rate in Neural Network - Geeks for geeks 06. November, 2025
https://www.geeksforgeeks.org/machine-learning/impact-of-learning-rate-on-a-model/