INF-2200-1 22H: Computer architecture and organization

Assignment 3

Adrian Moen, Amund Strøm

UiT id: amo172@uit.no & ast283@uit.no

GitHub users: AdrianMoen & ast283

GitHub Classroom: assignment-3-amund

November 11, 2022

1 Introduction

This report describes the design and implementation of a cache simulator for a memory system and the testing done to get the best hit/miss ratio on the simulator. The cache simulator consists of 3 different and unique caches, a level-1 read-only instruction cache, a level-1 read-write data cache and a level-2 read/write unified cache.

2 Methods

2.1 Design

A cache is a temporary memory that lets the CPU access data significantly faster compare to accessing data from the main memory. The reasoning for this is that the cache is much closer to the CPU and much smaller which makes it faster to access. As mentioned it is a temporary memory which means that only stores the recently used data, and quickly throws that out of the storage if it is not used again. This is because if the CPU just used some data, its highly likely that it needs the same data in the near future.

As specified in the introduction, our cache simulator have 3 unique caches and is 2 levels deep. The level-1 instruction cache should be read-only, this means that the data that is stored in the cache cannot be modified during runtime. The opposite can be said about the level-1 data cache and level-2 unified cache that is read-write, this means that data can be read, modified, added and deleted during runtime.

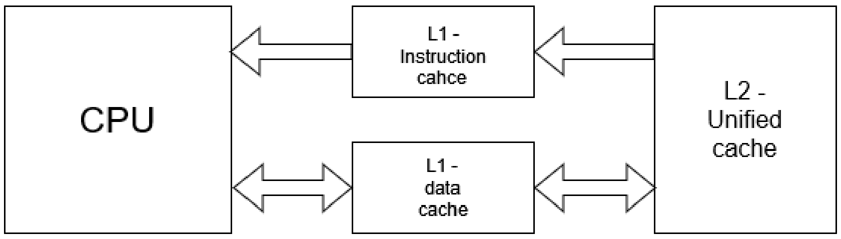

Figure 1: An illustration of the cache simulator. The arrows indicate the flow of data.

The cache have a set of parameters that decide how the cache should operate. Firstly there are the number specific parameters such as the size of the cache and the size of a single block in the cache. These determine the shape of the cache and how the cache is supposed to divide the CPU address it gets as input into three different fields. The fields are (form the highest order bits to the lowest), the tag which is used to translate a cache address to a unique CPU address, the index which is used to access different sets of the cache, and the offset which tells what data the CPU wants (we don't use this in this simulator).

The second type of parameters decide how the cache operate. These are called the associative of the cache, the replacement-policies and the write-policies. The associative divides the blocks of a cache into different sets or indexes. For example a direct mapped associative will create equal amount of sets as blocks, meaning at index x, there is only one block to hold data. On the other hand we could have a 8-way associative which gives us 8 blocks to hold data at index x. There are different pros and cons for all the different associativities that will affect the hit/miss ratio of a cache.

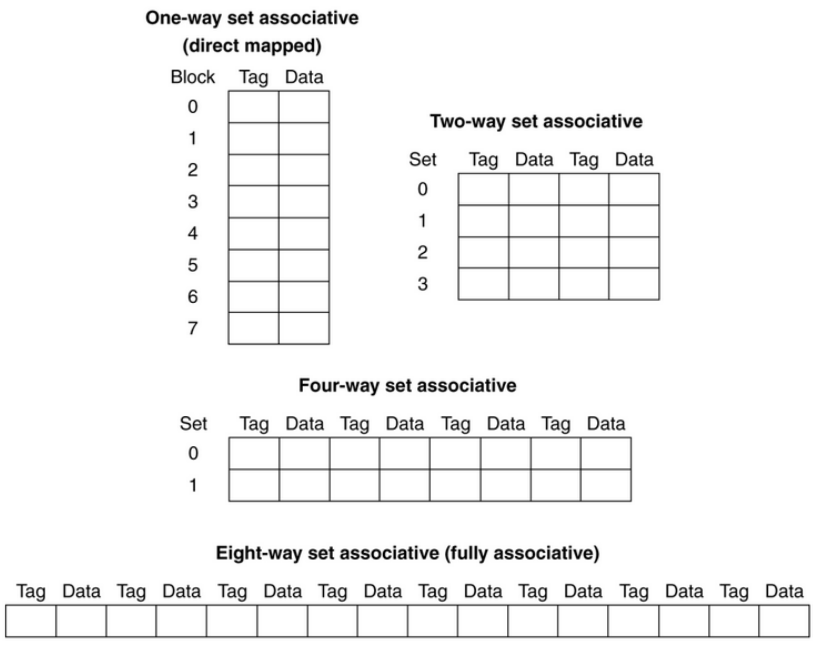
One-way set associative
(direct mapped)
Block Tag Data
0 1 2 3 4 5 6 7
Two-way set associative
Set Tag Data Tag Data
0 1 2 3
Four-way set associative
Set Tag Data Tag Data Tag Data Tag Data
0 1
Eight-way set associative (fully associative)
Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data

Figure 2: An illustration of the different cache associativities.

The replacement policies tells the cache which block of data to overwrite when the cache is full. Lest say the replacement policy is Least Recently Used (LRU) and the cache is about to replace an element inside a full set/index, the cache will then write at the block which is least recently used. These will affect the hit/miss ratio of the cache, depending on what kind of program that is being executed.

The write policies tells the cache in which level to write data. The most common policies are write-through and write-back. Write-through tells the cache to write to all the different levels when a write is done. Write-back tells the cache to only write to level 1, but when the data that is in level 1 is to be replaced, it will write the replaced data into level 2 and so on. This also affect the hit/miss ratio of the cache, but it will also affect the speed of the CPU since less writes and checks to be done on level 2 the faster the CPU can work since it wont have to wait for the operation to complete.

2.2 Implementation

How well does our implementation match the overarching design of the cache memory hierarchy? We tried simulating a cache in such a way that it would operate as close to a real cache as possible, yet the only important part is simulating cache hits or
misses. For this, we have utilized structs heavily to implement the design. The cache consists of three individual
cache$_t structs.Insidewhichwefindapointertoastructcontainingalltheparametersofthecache.Thisisinessencenotnecessary,$
Inside the
cache$_t structyouwillalsofinda3dmatrixofblock_t structs.Theseblockstructsrepresentasinglecachelineorcacheblockinsideallt$
We have also faced a lot of difficulties during our time implementing the cache. One in particular was how we would represent the cache sets and blocks inside the cache struct. At first, the only feasable idea appeared to be using sets structs that point to each other, sort of like a linked list. Inside these would be block structs that also point to each other. Then it became simply arrays of blocks that point to each other. The issue was how, and what we needed to allocate for a 2d matrix to work, and if it would be a double or triple pointer. We

solved this issue after a while, and used triple pointers. Ranking the blocks for eviction was another thing. We found out they would need to be ranked differently based on eviction policy. To keep this as dynamically as possible we had to figure out how to use pointer functions to store function calls inside a struct. Both eviction and write policies are represented this way.

2.3 Experiments

During this assignment we aim to test the different configurations of the cache. We change the cache parameters to whichever configuration we aim to test, run the simulations, and compare the hit to miss ratio of each configuration. Actual cache speed, and how long we spend storing/retrieving data is not measured, so results might be somewhat skewed towards favouring a write through due to its inherit extra time consumption for its write policy.

3 Results

Since there are so many different variations of cache, we're going to be limiting parameter changes to only associativity and policies, focusing mainly on policies. We did try with different configurations to see the relative performance of the different associativities. For instruction, data and unified cache respectively, we got these scores (same associativity for all):

Test	fully	direct	two way	four way	eight way
instruction	99.77	93.44	97.93	99.75	99.72
data	98.90	93.31	97.75	98.04	98.52
unified	51.28	65.83	85.63	74.10	71.58

Table 1: Hit rate in percent, LRU with write through, same associativity for all

As we can see from the results, the hit rate for every associativity is fairly high, except for direct mapping, which we can see lags behind the others. For the next test, we're going to be testing out eviction and write policies. To keep it simple, we are using the other parameters specified in the provided readme, and switching out the write policy for each eviction policy

Write Through	LRU	LFU	FIFO	RND
instruction	99.75	91.78	99.74	99.72
data	98.52	79.71	98.22	98.25
unified	70.61	54.68	72.75	66.05
Write Back	LRU	LFU	FIFO	RND
instruction	99.75	91.78	99.74	99.72
data	98.52	79.7	98.23	98.27
unified	70.60	54.7	72.75	65.96

Table 2: Hit rate in percent, alternating policies, default parameters from readme

As we can see from the table, the l2 cache has a lower hit rate than the different l1 caches, and we can also see that there is hardly any difference from write through and write back. If we where to measure time spent. perhaps we would see some differences. We can also see that RND seems to be a perfectly good candidate under certain circumstances, even beating LFU on all the points. One thing to keep in mind, is that we have only run the RND eviction policy once, and yet we have a certain consistency in the results. Going by this data, the conclusion that both FIFO and LRU seems to performing the best in this scenario, might not be wrong.

4 Discussion

4.1 Cache indexing

One thing regarding our specific implementation that can be discussed is the indexing of a cache set. During our implementation, and trying out the linked structs. We had to think how the cache is actually index in the physical computer. If it turns out that set lookup is almost instant, then we currently have a close representation to such a system. If it turns out however, that cache sets are actually iterated over, and compared for each time to find the right index, then a linked struct implementation would make more sense. We have currently not found any concrete information on this.

4.2 Collision

Since we are dealing with 3 separate caches, one of which is a unified cache containing information from both, there is a risk of something called collision. This happens when we access data for a certain address, find the data in the l2 unified cache, but it turns out that it is not data from the data memory. Instead it is an instruction from the l1 instruction cache. We currently have no way of handling this issue, and we have been advised to look past it. A simple solution would be to add another check for each block to see from which cache data was last stored, similar to a dirty and valid bit.

4.3 Conclusion

In this assignment we have successfully implemented and simulated a cache memory hierarchy, and done testing with the different configurations a cache can have, to see which one would fare best. We have concluded that both LRU and FIFO fare the best given this scenario.

5 Sources

References

[1] Tanenbaum, Andrew S. Modern Operating Systems, 4th edition. Pearson, 2015. Chapters 3.3-3.4.