TÉCNICO LISBOA

Lab02 Report- G001

(ist1102082, ist102663, ist1103252)

OC Lab02 Report:

In this project, we were challenged to implement a cache system by including features we learned in class. The available source code was programmed to only sustain the capabilities of a single lined, direct mapped cache. We were then given 3 tasks. In this report we will go through all these assigned tasks, depicting the major alterations done from the source code while explaining our thought process.

Task 01 - Directly Mapped L1 Cache

In this first task we were asked to modify the source code in order to implement a first level Cache (L1 Cache) with direct mapping and multiple lines. This means that each block of main memory will only be able to be stored in one specific cache location.

We started by removing the Cache data vector and moved it into the cache line struct in *Cache.h*. In the '*accessL1()*' function we started by initialising all the cache and setting all of its lines to invalid, as their content is still meaningless. Because we are now dealing with multiple cache lines, we need to deconstruct the address of the memory block in order to obtain the location where it should be stored in our cache. Since we are dealing with a block size of 16 words and each word is 4 bytes, each cache line will have 64 bytes, meaning that we will need 6 bits for the offset, so that we can locate each byte in the line. Since the L1 cache was said to contain 256 lines (2^8) we will also need 8 bits to reference the cache line (index). The remaining bits will be seen as the Tag. << *L1Cache.c: 61-78*>>

In order to get the number of bits more efficiently, we decided to create a function, '*logBase2()*', to calculate the logarithm of base 2 of a given input number. <<*L1Cache.c: 32-44*>>

After getting the cache line we will then check if our access is a 'miss', by checking if the line is invalid or if there is a Tag mismatch. If the line's dirty bit is set to 1 it means that we have changed the content of the line, thus we must get the desired data block from the RAM and write the one that's in cache in said DRAM. << *L1Cache.c: 83-96*>>

After dealing with the miss, we must analyse which type of access we are trying to make. If the 'mode' variable is *MODE_READ*, we are trying to read from the L1 cache, thus we copy the cache line content to the vector 'data'. If the mode is set to *MODE_WRITE* we are trying to write on the line, so we write the content of the 'data' vector on the said line and set the dirty bit to 1. << *L1Cache.c: 98-107*>>

With these functionalities, we have successfully implemented a directly mapped L1 cache.

Task 02 - Directly Mapped L2 Cache

At this stage we are being asked to, on top of the L1 cache, implement a L2 Cache with direct mapping. Since we now have 2 caches, in order to simplify our code, we decided to create a struct *'cache'* that stores both L1 and L2 caches. <<*L2Cache.h: 49-52*>>

The '*accessL1()*' function pretty much stayed the same as in the previous task. The only noteworthy change is the fact that, in case of a 'miss' the program will no longer access the DRAM, and it will look for the data block on the L2 cache, by calling the '*accessL2()*' function. If we are in the presence of a L1 Cache dirty block that needs to be changed by another, the program will write it

on the L2 Cache. We will only write the data in the DRAM when we have to substitute a dirty block on the L2. *<<L2Cache.c: 86-89>>*

This function (‘*accessL2()*’) follows the same format as the ‘*accessL1()*’ function from the first task. The main difference resides in the way we divide our addresses. The number of offset bits will stay the same as the block size has remained unchanged (6 bits). Since we now have a larger number of cache lines (512 as defined in ‘*Cache.h*’) we will need one extra bit (9 in total) for the line index. *<<L2Cache.c: 123-140>>*

Thus, we have now implemented a L2 cache.

Task 03 - 2-Way L2 Cache

In this third and final task we were asked to change what was done on the previous task and turn our direct map L2 cache into a 2-way associative cache. This means that now in our L2 cache each memory block will have 2 different locations where it can be stored. Such change will increase the size of each cache line (twice the block size = 2*64 = 128 bytes) but at the same time the number of lines has been reduced to half (256). *<<Cache.h: 11>>*

Since its L2 cache line will have two blocks, we have created a new struct ‘*CacheL2Line*’ that holds 2 dirty, tag and valid bits one for each block in the line. Also, we have added a new control bit, ‘*Time’*, for each block that lets us know which of the two data blocks has been the least recently used (LRU). By setting the ‘*Time’* bit to 1 we are indicating that such a block is the one that is going to be replaced next time there is a need to replace a block in its cache line. To accommodate this associativity change we doubled the data array size of each line so that it now holds two blocks. *<<L2W2Cache.h: 38-44>>*

The ‘*accessL2()’* function has been changed once again. When we initialise the L2 cache we not only have to set all blocks to invalid as we also have to set, per cache line, the first of its blocks’ ‘Time’ bit to 1 (so that we start by writing on the line’s first block) and the other to 0. It is also important noting that while we decompose the address that the L2 cache has now 256 lines the number of bits needed to reference the index will be 8. *<<L2W2Cache.c: 115-125 and 134>>*

After getting the target cache line, we need to iterate through the 2 data blocks in order to find out if one of them is the one we are searching for. It if was not found, a cache miss has occurred. It is then needed to check which block has been least recently used to substitute it. If it is dirty, we write its content on DRAM, like before. Every time L2 cache is either read or written, we must update the ‘Time’ bits, so that the block we are accessing is classified as the most recently used in its cache line. *<< L1Cache.c: 150-213>>*

We have now implemented the 2-Way L2 cache and finished all assigned tasks that allowed us to comprehend and deepen our understanding on how caches work.