CSCE 312: Computer Organization - Final Project

Texas A&M University, Spring 2020

Date: 04/30/2020

Student 1 name: Dominic Rivas **Student 2 name: Q-Dropped**	**Student 1 UIN: 326002950** **Student 2 UIN: Q-Dropped**

Introduction

This report discusses a simulation of how cache is handled within a computer system. The following program uses a driver file and a Cache class to simulate a computer's cache. The cache class was constructed in the form of a chained hash table. The structure consisted of two structs, the Set and the Blocks contained within the set. The RAM is held in the main.cpp for simplicity and is constructed as a map.

Design and implementation

In this project, first the RAM is initialized into a map<string,string> object using the input.txt file. After, the user is prompted to configure the cache and then the cache is initialized with an insert function parameterized with an index for the set, a tag 00 for the block and a vector of 00's as values and constructs a certain number of sets depending on the cache size divided by block size and associativity. Then the program constructs a number of Blocks within each set, relying upon the number of the associativity. After the RAM and cache is done being initialized, a menu of possible commands pops up.

Cache-read && cache-write

Cache-read was implemented to first convert the given address into relevant data such as *offset, tag, and index*. Then the function checks for a hit with the *tag* while also incrementing all of the *used* parameter of a block by 1 and resets the *used* parameter to 0 if accessed through hit or through a miss. The used parameter is for the least recently used policy. The larger *used* is, the longer the block has not been used. The parameter *frequncy* is only incremented when the block is access by a hit or a miss. The smaller *frequncy* is, the less it has been accessed and thus targeted by the least frequent policy.

Cache-write was constructed the same exact way as cache-read with only a few modifications to accommodate the write policies by adding corresponding if-statements to the period of data modification which is after the replacement policies.

Cache-flush && cache-view && memory-view

Cache-flush's *makeEmpty* and cache-view's *printMap* were both implemented recursively with nullptr being the base case in each of the functions. In cache-flush, before the cache is cleared, the program calls on the function *check_dirty* which updates the RAM with Block's data where it's dirty bit is 1. Then flush calls on *makeEmpty*, the root is skipped over to allow reinitialization of the cache, thus clearing the cache.

Cache-view outputs the private members of the cache when at the root and then proceeds to output each blocks' private members. Memory-view was just iterated on with a for-loop

Cache-dump && memory-dump

Cache-dump just iterated over the cache with a nested while loop and outputted the data into the cache.txt file .Memory-dump was iterated over with a for-loop and just pushed the data into the ram.txt file.

Conclusion

Overall, the program was pretty fun constructing as a class. One major thing I need to work on with this code is reducing the number of imports to the program as I only used only one or a few functions of all the possible functions. One main issue that I still cannot figure out is why my destructor destructs the cache in the middle of the program. Besides that case, I found exploring the usage of this data structure entertaining and better helped me visualize the cache.