CISC 340 – Cache Simulator – Overview

James Allender
Vincent Li

December 10, 2018

1 Description:

The Cache Simulator project simulates how a cache works in a computer system through intercepting the processor's memory requests and handling the retrieving and storing to and from memory. Our project is based off of the single cycle behavioral simulator from project 1 and is specifically based on the professor provided sim.c file. The program consists of a single procedural C program that reads in a given file containing machine code and simulates the machine code to demonstrate the cache behavior. The program prints out the cache operations that are taking place in the system to the screen.

2 Overview:

Our project can be built by running the included make file:
$make
Our project is run using the syntax:
$./cacheSim -f [machine code file] -b [words per block] -s [num sets] -a [associativity]

Our cache system has a cache size equal to (words per block * number of sets * associativity) where these values are provided by the user. The level of associativity ranges from 1 (direct-mapped cache) to the number of blocks in the cache (fully associative cache). The block size is the number of words of memory that are stored in a single cache block. Our write policy is allocate-on-write write-back where we get information from memory if it is not in cache already and we write data back to memory from cache if it is dirty when it is evicted. When chosing what block to evict we prioritize any invalid block of memory in an appropriate set and secondly the least recently used (LRU) block of the appropriate cache set.

To handle read requests from the processor, our program will first check if the the block of data for the request is in the cache. It does this by searching through the ways of the cache set that corresponds to the desired address for the address's tag and that that block is valid. If the address is in cache and valid, we get the desired block offset for the desired word of memory in the block from the address and transfer that word to the processor. If it doesn't find the address in cache it transfers the block of data containing that address from memory to the cache. Once we have it in the cache we transfer it to the processor as previously described. When deciding where to put the block from memory in cache we look in the set corresponding to the address for a way with an invalid block or the block of the LRU as described above. After determining the cache line to allocate, the program will overwrite that way with the data and transfer the data to the processor. However, if the data of the LRU is dirty it will be written back to memory before being reallocated.

To handle write requests from the processor, the program operates similar to how it handles read requests. The program first finds if the address is in the cache. If it is in the cache, it changes the word of memory in the block corresponding to the address with the new data attempting to be written by the processor. If it is not in the cache, the program will transfer the block of data from the memory to the cache the same as it did for a miss read request, then it writes the new data to the block in the cache but does not immediately write it to memory. This is in line with our allocate-on-write write-back policy. After a cache entry is written to, the dirty bit is set indicating the data will need to be written back to memory. The same as with read, if the program is going to evict a dirty block of memory to take in a needed block for writing, it will write this block back to memory before reallocating the line.

At the end of a program when a halt is encountered a halt request is made to the cache system. The cache system then loops through all of the cache entries checking for valid dirty cache blocks. All of the found valid dirty cache blocks are written back to memory in order to maintain data correctness. After the blocks are written back they are all invalidated. This ensures all dirty data makes it back to memory after a program finishes execution and invalidates all valid blocks to ensure program correctness for when the next program to run.

3 Challenges:

The biggest challenge we had in this project was in understanding exactly how the cache interacted with the CPU and the system. Specifically our issues revolved around how to design our program to properly execute the cache operations. our design started as a single function that took in memory requests and either pulled from memory into cache and returned that, or put into cache and stored that to memory. This design evolved into 11 separate functions that handle different aspects of cache operations and are all coordinated by the **cacheSystem()** function.

When we started our design we underestimated the functionality the cache system was going to need implemented in order to accomplish its task. Part of this was all of the different operations that get repeated a few times in different cache operations. We ended up separating all this functionality into different functions to make it easier to follow our program and to reuse certain code in different locations. However a number of the functions are only used once for a specific cache operation, but it was still easier to follow the execution and troubleshot the program having the functionality segmented. For example **searchCache() alocateCacheLine()** and **incrementCyclesSinceLastUse()** are all functions that are used once, but they all accomplish specific tasks which are easier to think about as a single self contained function.

An area that also presented a significant challenge too us was understanding pointers in C and how structures and arrays actually work in the language and how to manipulate them. After extensive discussions with the professor we were able to understand to an adequate degree how arrays and structs variables work as pointers to a memory location and all how all accesses to these structures are just offsets the the base address the pointer points to. Understanding this allowed us to understand how C is just a thinly veiled interface with memory and how we could manipulate our cache information by a simple accessing memory rather than trying to rebuild and overwrite structures or arrays.