Programming Assignment 3: Cache

Edmond Wu, CS211, Section 02

December 15, 2014

READ ME

Design and Implementation:

Designing and implementing this program had essentially two parts: building the initial data structure, and then populating the cache using the contents from the trace file. The data structure I used to represent the cache was a multidimensional array, with a struct representing the cache. The cache contains an array of sets, and within each set there is an array of cache lines. Aside from storing the actual structures, the cache also stores the information of relevant variables, mainly the parameter arguments (such as cache size, cache block size, etc.) passed into the program by the user so that the program may use that information later if need be.

The second part of the program revolves around populating the cache with information. For a write-through cache, each line from the trace file is essentially stored in another struct which handles all information that could be retrieved. The memory address and its appropriate operation (read/write) is stored in the struct. The memory address is in hexadecimal format, which is then converted into binary. With that binary address, given the information passed by the program user, it calculates the necessary amount of bits required for the tag, set index, and block offset, and also stores those fields in separate fields. With all the necessary information gathered, the program then goes through the cache, using a decimal representation of the set index to find the corresponding set number, and then checking each line in that set for a validity of 1 and a tag match. If both match, a cache hit results, and depending on whether the operation is a read or a write, it updates the cache appropriately. If there is no match after going through every line in the set, a cache miss results and then it performs a main memory read and/or write (which simply means the memory read and/or write count get incremented in the context of this assignment) and stores the information from the address into a new cache block. It will look for the next empty cache line (with a valid bit of 0) and then give that cache line a tag and set its validity to 1.

If there are no empty lines in the cache set, then the LRU replacement algorithm is used (least recently used). To figure out which block is the least recently used, every cache line has a corresponding index value, which gets incremented every time a new cache line is loaded. The least recently used cache line happens to be the line with the highest index. Every time a cache block gets used (even if the cache is not full yet), its "LRU" index gets reset, which simply signifies that it has become the most recently used block. It does not matter whether the cache is direct, fully associative, or n-way associative as the only difference between those caches is the number of lines that go into a set. That number is determined by the user parameters, and the cache's corresponding arrays are set to match that number.

There aren't too many differences between a write-through and a write-back cache in the context of this assignment, but the main difference involves the use of a dirty bit (where the value of the bit is 1 if it's dirty and 0 if not). If a line is dirty, then a write to main memory occurs (in the assignment that translates to incrementing the memory write count). The dirty bit is made dirty every time a write operation occurs (both hit and miss), and is set clean if a read-miss occurs, while nothing happens if a read-hit occurs. Because of certain complications involving the dirty bit, I used 2 separate functions to differentiate between write-through and write-back caches.