Programming Assignment 4

by David DeSimone

May 5, 2014

1 Cache Implementation

I implemented the cache as a struct called "cache". A cache struct has a number of fields defining it's "type" (direct, assoc etc.) and a pointer to an array of it's sets. The sets themselves are structs, which are each only a pointer to an array of lines. A line has a valid bit, a dirty bit, a tag, and an age counter. Our simulation didn't require we actually PLACE anything into the block portions of the lines, so they were omitted to save memory. When the program starts, a cache is created to user specification by an initialization function. After this point, the program will read over every line of the input file and attempt to process the given memory address.

2 Write Policy

When the user inputs the write policy to the program, an int is set representing the write policy is set (0 for write back and 1 for write-through). When something is written to the cache, I check the dirty bit and write policy. If the dirty bit is set and its a write back cache, I increase the number of memory writes. If its a write through cache I increase the number of memory writes. When I write to the cache, I set the lines dirty bit (which makes the implementation differences between wb and wt minimal).

3 Rejection Implementation

To implement the **LRU** rejection policy, I created an atomic counter, called x. Whenever a cache hit/miss occurs, I increase x, and "stamp" x on that cache line. Meaning that each line struct has a field "y", which marks the counter when it was last used. When it is used again, the timer is updated. When a rejection is to occur in a set, I iterate over all of the line structs and see which one has the smallest y value. The line with the smallest y value is the one to be rejected.

4 Direct Map

To implement the direct map cache, I create a cache struct with a set array, but only create one line struct per set. When the program later receives an address to process, I isolate the set bits from the binary address, and find the corresponding set in the set array. From there I preform a tag comparison to determine if we have a cache hit or cache miss.

5 Fully Assoc Cache

To implement a fully assoc. cache, I do something similar to the direct map case. I initialize the cache, with only 1 set struct. I then create an array of line structs, and associate my singular set struct with the line array. When looking for a cache hit or miss, I iterate through the line array, comparing tag values. I use the aging mechanism described above to determine what line to reject when a miss occurs and every line is valid.

6 N-Way Assoc. Cache

To implement a N-way assoc. cache, I combined the two approaches of the previous two sections. I initialize the cache, with a set array of necessary size (cache size / B * N). From there I iterate over every set, and create a line array of size n.

When looking for cache hits, I check the addresses set bits to get the set array. I then examine said set, and iterate over all of its line, comparing tags. Again, I use the aging mechanism described above for rejection.