COMPUTER ARCHITECTURE (COL216)
ASSIGNMENT 12: SIMULATING CHACHE MEMORY

DIPEN KUMAR
2018CS50098

DESIGN –

I have used vector of vector of vector of unsigned long long int to create cache. I have also created main memory which is a vector of long long int. I have initialized my RAM with random values. My RAM and cache both are block addressable, which we were asked to implement for convenience, since the motive of this assignment is to simulate and observe the performance of our replacement policy on hit rate. Read, write in cache is at block granularity.

I read the input file to get the details of cache to create. I get cache size, block size, set associativity and the value T which is the number of accesses after which a block (cache line) is moved from high priority to low priority group. As we increase the value of T we will increase the number of blocks in the High-Priority group and decreasing T will decrease the size of High-Priority group and increase the size of Low-Priority group. Hence the question asked in problem statement- Are the High priority and Low priority groups fixed in sized? The answer to which is NO!. As we have seen these sizes depends on the value of T and input access pattern. Also another question asked was- Can these sizes change at runtime. YES! It depends on the input pattern. Suppose at some point during run time we have memory access pattern which accesses again all the cache lines of a set after the initial access that fetches it into the cache. This means every block of that set is now in High-Priority group making Low-Priority group empty. Now suppose at some other point during run time we have another pattern of memory access where no block from the same set is again accessed for sufficiently long T cache accesses. Here we will see that all cache lines of that set are in Low-Priority group and none are present in High-Priority group. Hence we saw that same set at two different points during rum time has two different distributions of blocks in high and low priority groups. Hence these sizes of High and Low priority groups change at runtime.

Now we will see why this T determines High and Low priority group break-up. If we increase T implies we are allowing the blocks which once has reached High-Priority group to remain there for a long time. Thus by increasing the time spent in High-priority group we are increasing the probability of finding a block in High-Priority group and other way. If probability of finding a block in High-Priority group is high implies High-Priority group has more blocks.

ASSUMPTIONS –

Assumed maximum block size be 8 bytes i.e. block data can fit in 64 bit long long int in C++. I have created my RAM (main memory) which stores 4096 blocks.

REPLACEMENT POLICY –

The replacement policy I followed in this assignment is –
In case of miss (read or write) I have first looked for an invalid position in the indexed set where I can place my requested data from RAM. If no such position found I have moved to Low-priority group to fetch the least recently used block in that group. If again suppose we have no element in that Low-Priority group which means Now we have no invalid positions and all are High-priority group member. Here we will look for least recently used block in high priority group. Once we have the target position to replace with the requested data. We will copy the data from main memory to that in indexed set in the cache. Also note I am using write back strategy which means before deleting the data that was replaced in the cache, I will check its dirty bit if it is on i.e. equals to 1 then I will write back this data in RAM. Once miss is handled and data is replaced we may now follow the normal execution steps i.e. we can read it from cache or write it in cache turning dirty bit on. I have followed write allot.

Now I will talk about the implementation of the replacement policy of the cache and my design discussed above in details with respect to time and space complexity.

IMPLEMENTATION –

In C++ I have traversed through the set and had maintained so far block index in the set to replace if incase miss occurs. That is I have given first importance to invalid position, I if got one then I am ignoring High-Priority and Low-priority group block. If I got Low priority block then I am ignoring High priority group block while eagerly waiting for invalid position to occur and I can update my index to replace also I am updating my low-priority block to another low priority block which was used more before. Lastly If I have high-priority block to replace I am updating it with another high priority block will lesser recent used and also seeking to switch to low-priority group or invalid position if I encountered any. Hence over all complexity of finding such block in the set to replace is O(n) where n is the associativity of the set. Also meanwhile during my search if I encountered the block whose tag matches requested tag. I stop and no need to replace since it is hit. When a block needs to replace I checked for its dirty bit. If it was on then I write it back to memory which in hardware is expensive and take more clock cycles and result in miss penalty together with the cost of copying the requested data from the memory to cache. In here in my software implementation, I have done in constant time i.e. O(1) to make my simulator efficient while mimicking the actual behaviour of hardware. Initializing my RAM took O(m) time where m is the size of my main memory. And Initializing the cache took O(l) where l is the size of the cache. Printing cache took O(l) and reading input memory access pattern from file took O(k) where k is the size of the input file.

Hence My implementation of this assignment is linear in time complexity with respect to size of set, size of main memory, size of cache and size of input testcase.
In terms of space complexity I have created a vector for main memory and for cache. Hence space complexity of my implementation is linear w.r.t size of main memory and RAM i.e. O(p) + O(q) where p is size of RAM and q is size of cache.

TESTCASES –

I have written a programme that i.e. TestCase.cpp which when given cache-size, block-size, associativity and number of memory access we want to simulate in Cache.cpp will generate input memory access pattern. This pattern is not completely random. Before I go further describing how I am generating input memory access pattern. I want to discuss the nature of our real program on computer. Our program generate memory request which show temporal and spatial locality. Now keeping this in my mind I have programmed my test case generator which generates input memory access pattern having temporal and spatial locality. This degree of spatial and temporal locality in generated input memory access pattern depends on the input given by user. I call this as degree of temporal and spatial locality in program. This value is between 0 and 1 both inclusive. When degree is 0, we see lowest hit rate because every time we generate random block address in input pattern to access. And this doesn't do anything to include temporal and spatial locality in input memory access pattern. But when we keep increasing this degree we see increase in hit rate because degree is the probability that next access in input memory access pattern will have same tag. This implies by increasing the degree we are increasing the probability that next block have same tag as the previous one but have random set-index which is greater than equal to 0 but less than number of set. If the random index-value matches the previous one that means it is the same previous address i.e. there is temporal locality in program. If the set-index is different from previous but I have already mentioned that it will have same tag value to that of previous. This implies the current address is near to previous address and the maximum distance between them is the maximum difference in their set-index which is equal to number of sets. Hence we have successfully included spatial locality in our input memory access pattern. When degree is one we always have next access which has same tag to that of previous access i.e. every access have same tag. Hence we will only have compulsory miss also called cold-start miss. These are cache misses caused by the first access to a block that has never been in the cache. Once it gets in the cache also we know that we have input pattern with same tag when degree of temporal and spatial locality is 1. This implies that next time if we get the set-index then we are going to search for the tag value which is same for all input and hence we will find that in set. Also in this case we don't need associativity of set, since there is no two or more blocks competing for same set-index because when degree is 1 we have same tag for all input and hence we have no two input competing for same set-index, no two different tag value competing for same set-index. Suppose if there is two input competing for same index also both have same tag because degree is 1. Hence both addresses are identical because both have same tag and set-index.

HOW TO RUN –

1. Unzip "2018CS50098.zip" to get a folder named "2018CS50098".
2. Open terminal and change your directory to "2018CS50098".
3. Run "make" command.
4. Entry input test-case specifications and press enter.

NOTE – First TestCase.cpp will run and using input provided by user through console will create input.txt. Then Cache.cpp will run on input.txt and provide detail simulation in output.txt and it will also display the final state of the cache and its statistics on console.