Computer Architecture Lab Assignment 7

Divya Rajparia ES22BTECH11013
Ahmik Virani ES22BTECH11001

Cache Simulator

November 12, 2024

This report has 2 main sections:

1. **Program Design:** Which explains our high level approach to building a cache simulator.
2. **Assumptions and Testing:** Which explains some assumptions we made, and the steps we took to check correctness of our code.

   Our code supports all 3 parts asked in the assignment.

Program Design

Please note, **we have extended our previous code of simulator by making 3 main changes**:

1. **Read and Write access functions:** We replaced direct reads and writes into memory by defining 2 functions - read_access() and write_access(). Now, every read and write access is directed to this function. If the cache is enabled, these accesses first look in the cache, and follow the procedure for hits/misses(explained below). If cache is not enabled, these functions simply access te memory directly.
2. Update queue: We have maintained a queue of the blocks currently in cache for each set/index. The block at the top of the queue for a particular index is the one that will be evicted if needed. We have implemented all the 3 asked replacement policies in this function itself
3. The other set o changes we made were in the main function. These were mainly to add support to the required cache commands, and maintain the different files asked.

Read/Write access functions

This function is only entered if the cache is enabled. This is the flow followed by both these functions:

1. We first find the offset, index, and tag of the requested memory location by using bit masks

2. In any access, we first look for the memory location in the cache. If it is found, we proceed with the read/write.

3. If we do not find it in the cache, the read and write accesses take different paths:

   (a) In the read access, we first look if there is an invalid bit in the cache. If yes, we get the entire memory block to the cache.

   (b) In the case of write access, if we succeed in finding an invalid block, if it is write back, we try to get the entire block, and if it is write through, we just directly write into the memory.

4. The last case is when we neither find the block in the cache, nor do we have an invalid block. In this case, we need to evict the block at the top of the update queue. If it is write back, we get the entire new block into cache, and if it is write through, we just directly write into the memory, without evicting anything.

Update Queue Function

As mentioned above, this function mainrtains a queue for every index of the cache, and the block at the top of the qeueue corresponds to the one which will be evicted, if need arises. This function has a parameter "to_replace", which specifies if the current request to this function requires a block to be evicted, or does it just need a block to be rearranged in the queue. In this function, we have 3 main parts:

1. FIFO: Here, if it is not replace, it means that we either found the block in cache, or found an invalid entry. In the latter case, we simply add the new block to the back of the queue. In the case of replace, we evict the queue top, and add the new block at the back.

2. LRU: This implementation is exactly like LIFO, except that - when a block is accessed, we bring it to the queue back - indicating that it is the most recently accessed block.

3. Random: In random, we don't really need a queue, we just work with the "to_replace" variable. If thre is something to be replaced, we randomly pick a number in the range of the index blocks, and evict it.

We have taken care to write back any dirty blocks into the memory in case of evictions.

Testing

We conducted extensive testing of our code, mainly on the 3 Homework 4 cache example codes. We tried varying all the variables, like block size, associativity, cache size, write policy, as well as replacement policy. We have attached those 3 code files for your easy reference.