Assignment 12

This report comprises the implementational details for the Cache Simulator in C++. Structures are used for representing the Block and an element of Cache i.e a set. The struct *Block* contains the fields namely valid (int), tag (int), data (int), and dirty (int). The struct *CacheE* contains the fields NS (int), which denotes the number of sets, *HP* (map <int, Block>), and *LP* (map<int, Block>) which correspond to the high priority and low priority blocks within a set. The **key** of the map is the **last instruction index** which used the block. A cache would be an array of CacheE elements whose size is calculated as:

Size = Cache Size (in bytes, Total) / (Block Size * Associativity).

The entire simulation process is implemented using 6 different functions. Namely,

- Init
- Search
- Fetch
- Update
- Insert
- Check

The detailed descriptions of these functions are as follows:

1. **Init:** This function initializes the array of CacheE elements by assigning the value NS equal to the associativity of the Cache.

2. **Search:** This function checks whether the queried address is present in the Cache or not. This is achieved by iterating over the blocks of priority line, followed by the low priority line. The check is made for the Valid bit as well. The function search is called for both, read and write instructions, so if for instance, a block in low priority is queried again and its key is within the

range of {current_index - T, current_index}, then it would be promoted to the high priority line and removing the corresponding block present in the low priority line. The function returns a boolean indicating whether the desired block is present or not.

3. **Fetch:** This function is called during a **read-miss**. This function fetches the value stored in the memory and inserts the block in the low priority line. If there is an existing block at the low priority line, then the dirty status is checked. If it turns out to be 1, then the value stored in the memory is updated, before the removal of the key from the low priority line.

4. **Update:** This function is called during a **write-hit** to update the value of the block in the cache memory. The new block is first found by iterating over the High and Low priority lines and the block is identified using the tag data. A new block is again inserted with the dirty field set to 1, and the data is updated, rest all the fields remaining unchanged. The new key would be the current instruction index which is passed as an argument to the function.

5. **Insert:** This function is called when a **write-miss** occurs. In this case, the block isn't present in the cache for updating the data value. In such a scenario, ideally, the block should be first read from the memory and then the value should be updated. However, for simplifying the implementation, a new block is directly inserted into the low priority block with its dirty field set to 1. Any other block already present at the location would get updated in the memory as well if its dirty field is 1.

6. **Check:** This function performs an iteration over all the sets in the cache memory. For each set, firstly, an iteration is made over the high priority line followed by a low priority line. For the high priority line, if the key of any block is less than the current instruction - T, then that block is removed from the high priority line and again inserted into the low priority line. After doing this for all the potential exits, an iteration is performed over the low priority line. The total number of blocks which can be present in both low

and high priority lines is constant, therefore the iteration is performed for removing all the excess blocks in the low priority line. The removal is done according to the LRU policy, and since maps store their keys in sorted order, no external sorting is required. This function is called at the end of processing every instruction and thus, the inherent state of the cache is conserved according to the initial constraints provided.

Remarks:

- The implementation allows variable sizes for the low and high priority lines, however, their sum i.e. total size would remain constant.
- For the case when associativity equal to 1, the line would be labeled either high or low, depending on the instructions.
- The output comprises the index of the cache along with the label for all the lines in addition to the output provided in the specification. The label for any line within a set can be:
    - H: denoting high priority line.
    - L: denoting low priority line.
    - U: denoting the unused line. These are the lines that were never involved in any instruction.
- The instruction processing step involves splitting the input in a line from the test case file. This is done until the end of the file is reached.
- For each instruction depending on whether the instruction is read or write the above functions are used.
- For read, the block is initially searched. If present the variable reads (denoting the number of read instructions) is updated
- Likewise, for write, we first search for the block in the cache. If the block is present, then we update the data and the key, and the dirty field. If the block isn't present, we simply insert the block to the cache.
- A particular **drawback** of the design is that if all the blocks are High priority within a set, then no new High priority instruction can be inserted until the T cycle condition holds.
- The implementation is tested against several test cases with different associativities and sizes of cache.

Param Khakhar
2018CS10362