COL216 Major

Vishal Bindal
2018CS50425

To Run:

1. To print the cache status after each cache access, as well as all main memory accesses, set `show_details = true` in main.cpp. Else set it to false, and only the cache status at the end, along with results, will be printed.
2. In case of show_details = true, in order to wait for a character input after each access, set `wait = true` in main.cpp. Else set it to false, and everything will be printed together.
3. Run ‘ `make` ’ to create an executable ‘main’
4. Run ‘ `./main` ’ if the input is from ‘input.txt’ in the same directory, or else ‘ `./main <input_file_path>` ’.

Design: Structure

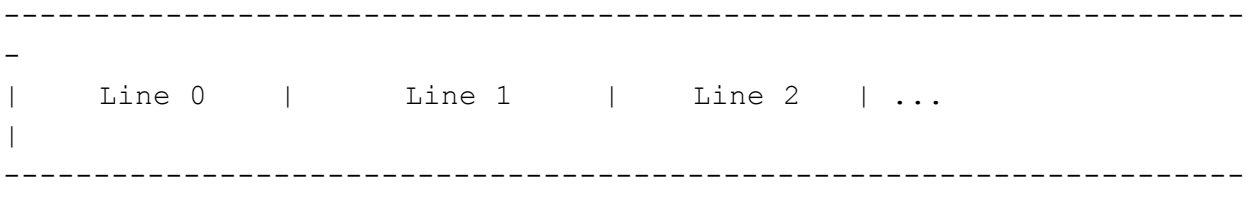
```
Each ‘Line’ in the cache has the following structure:
-------------------------------------------------------------------
-Valid | HPG | Dirty |      Tag     |     Data    | Access_order |
TTL
-------------------------------------------------------------------
-

where
Valid (1 bit): valid bit
HPG (1 bit): true if the line belongs to the high priority group in
the set, false if it belongs to the low priority group
Dirty (1 bit): dirty status
Tag ((mm_address_bits - log(no_of_sets -1)) bits): Tag Here
mm_address_bits = no of bits in any address to main memory,
no_of_sets = no of sets in the cache
Data (8*size_block bits): Here size_block is given in bytes
```

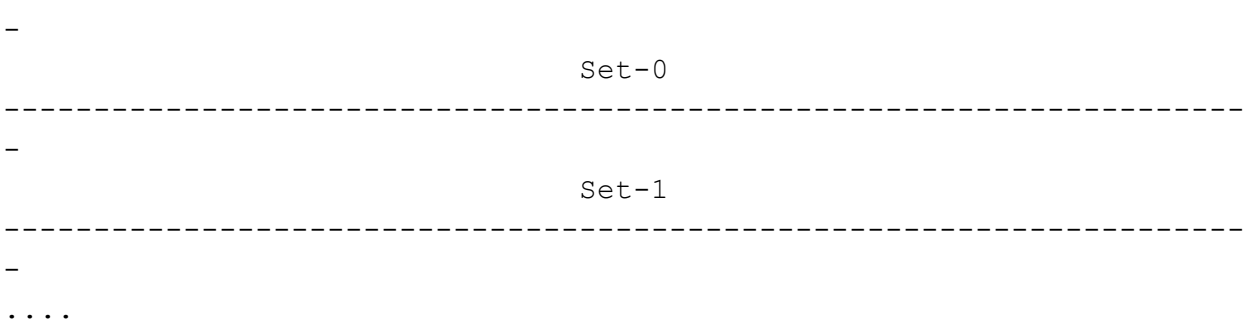
Access_order *(log(associativity) bits)*: It denotes the order in which the lines in the set are accessed. access_order = 1 denotes most recently accessed. For associativity=4, access_order = 4 denotes being least recently accessed.
TTL *(log(T) bits)*: TTL denotes time-to-live for lines in high priority blocks. When a line is promoted to HPG, its TTL field is updated to (T-1). For each subsequent access, if the line is not accessed, this TTL field is decremented by 1. At the end of Tth access, when the value of this field is zero, the HPG bit is set to 0, denoting demotion to low priority group.

All the above fields are implemented as bool or vector<bool> accordingly.

Each **'Set'** is a vector of Lines.



The **'Cache'** is a vector of Sets.



The **'Main Memory'** is a vector of blocks, where each block is a vector of bits
Size : mem_size X (block_size*8) bits = 2^16 bits
The total bits allocated to the main memory are kept constant at 2^16. Mem_size (which is the no. of blocks in the memory) is calculated using block_size (given in bytes)
Range of valid addresses: [0, mem_size - 1]

Other considerations and assumptions:

- All addresses are considered as block addressable, as mentioned on Piazza. i.e. The address doesn't contain offset bits.
- Main memory is block addressable, i.e. each address of the main memory points to a block, instead of a byte or word.
- Data, though allocated block_size*8 bits, should be less than or equal to 8 bytes in size. This is because long long has been used while parsing the input file, which is then converted to vector<bool>. Upper limit of 8 bytes was also mentioned on Piazza
- Data is signed.

Design: Priority groups and replacement policy

The priority groups can change at runtime, since each line has a bit to denote low or high priority.

HPG bit control:
In case of a read/write **miss**, when a new block is added to the cache, its priority is set to low by keeping HPG bit as 0.
In case of a read/write **hit**, the HPG bit is set to 1, and TTL set to (T-1). In each subsequent access, if this block is not accessed, the TTL value is decremented by 1.
In case it is accessed again while in high priority, it's a hit and hence TTL is bumped up to (T-1)
If TTL is zero and hence cannot be decremented any further, it is shifted to low priority by setting HPG as 0

Effect on replacement policy:
If a set is full and a miss occurs, then a best position for replacement has to be identified.
The replacement policy, in decreasing order of priority, is:
1) Valid bit zero
2) HPG = 0, and minimum access_order
3) HPG = 1, and minimum access_order

Design choice justification:
Setting an HPG bit allows no of blocks in high priority to lie in the range [0, associativity].

This is a better choice than fixing no. of blocks in high priority to a constant, since that would lead to wastage of space in the set in these cases:

1. If no blocks, or very few blocks, are accessed again while they are in the cache, space allocated to high priority blocks would be wasted
2. If a lot of blocks are accessed again, then the space for high priority would be less, hence some of them would have to be replaced.

Dynamically changing no of high priority blocks takes care of both these issues.

Justification for access_order

This is required for the LRU policy for each group in a set.

1. One choice would've been to sort blocks in a set instead of storing access_order bits. However, this would've led to shifting of whole lines (including data, tag bits) on every access, which is expensive. Updating one field in every line is more efficient.
2. access_order is kept independent of high and low priority groups. This is because blocks can switch between the 2 groups, so their access order relative to other blocks in the set should always be preserved.

Algorithm

1. The 4 parameters (cache size, block size, associativity, T) are read from the input file, and cache and mainMemory objects are initialised
2. Access instructions are read from the input file. Address and data values are first converted to long long, and then to vector<bool>. Cache read or write function is called accordingly.

3.1. Cache Read

A. The set is identified by reading the bits of address between set_bits_start and set_bits_end (initialised in step 1). Tag is then the rest of the address.
B. Search for a hit: Line for which valid = 1, and tag = query tag
C. In case of a hit:
    a. set is_high_priority = 1, to shift to HPG
    b. Set TTL to T. (changed to T-1 at end of cycle)

c. Update access_order for all lines in the set, by incrementing access_order of valid lines which was less than previous access_order of the hit line
d. Set access_order for this line to 1
e. Return the data

D. In case of a miss:
a. Get an appropriate position to insert the new block using the replacement policy. Let this be p.
b. If p's valid and dirty bits are set, then write back has to be first performed before fetching the required block.
i. Calculate main memory address by concatenating tag and set index.
ii. Write data to this main memory address
c. Read required data from main memory and update line's data field
d. Set valid=1, dirty=0, is_high_priority=0, TTL=0 and the tag field
e. Update access_order of all lines
i. If the selected position was valid before, then access_order of all other valid lines is simply incremented by 1.
ii. Else increment access_order of valid lines which were less than previous access_order of the p.
f. Set p's access_order to 1
g. Return the data

E. Update groups for all lines in the cache
a. If the line is valid and in HPG, and TTL is non-zero, decrement TTL value
b. Else if the line is valid and in HPG, and TTL is zero, then set HPG to 0.

3.2. Cache Write

A. The set is identified by reading the bits of address between set_bits_start and set_bits_end (initialised in step 1). Tag is then the rest of the address.
B. Search for a hit: Line for which valid = 1, and tag = query tag
C. In case of a hit:
a. set is_high_priority = 1, to shift to HPG
b. Set TTL to T. (changed to T-1 at end of cycle)
c. Set dirty to 1
d. Update access_order for all lines in the set, by incrementing access_order of valid lines which was less than previous access_order of the hit line
e. Set access_order for this line to 1

f. Return the data

D. In case of a miss:
   a. Get an appropriate position to insert the new block using the replacement policy. Let this be p.
   b. If p's valid and dirty bits are set, then write back has to be first performed before fetching the required block.
      i. Calculate main memory address by concatenating tag and set index.
      ii. Write data to this main memory address
   c. Set valid=1, **dirty=1**, is_high_priority=0, TTL=0 and the tag and data fields
   d. Update access_order of all lines
      i. If the selected position was valid before, then access_order of all other valid lines is simply incremented by 1.
      ii. Else increment access_order of valid lines which were less than previous access_order of the p.
   e. Set p's access_order to 1
   f. Return the data

E. Update groups for all lines in the cache
   a. If the line is valid and in HPG, and TTL is non-zero, decrement TTL value
   b. Else if the line is valid and in HPG, and TTL is zero, then set HPG to 0.

Testing

The following test cases are attached:

1. 12 Test cases in **testcases.txt**
   a. 9 test cases to demonstrate correctness and working, and exceptions
   b. 3 test cases to study effect of changing associativity and T, on the hit ratio
2. 3 tests in **test.cpp**, where cache and mainMemory are initialised and following 3 tests are present to demonstrate correctness of cache:
   a. Calculating nth fibonacci number
   b. Calculating sum of first n natural numbers
   c. Quick sort of an array containing a random shuffle of numbers 1,2,...n

In all tests, read/write operations are done using cache's read and write functions, and the final answer is verified.

3. A large test case in **inp_qsort.txt**, which is generated using all read/write operations in a quick sort of a random shuffle of first 100 numbers.

Results

1) **Effect of cache size**

in input inp_qsort.txt
Block size = 8 bytes, associativity = 2, T = 5

Cache size (bytes)	Hit ratio
16	0.363
32	0.536
64	0.627
128	0.703
256	0.781
512	0.856
1024	0.949
2048	0.949

Thus, hit ratio increases with increase in cache size as expected, and saturates when all 100 elements of the array can be stored in the cache.

2) **Effect of associativity**

in input inp_qsort.txt
Cache size = 128 bytes, block size = 8 bytes, T = 5

Associativity	Hit ratio
1	0.701

2	0.703
4	0.691
8	0.685
16	0.684

Thus, for a large test case having mostly sequential accesses (elements of an array nearby are accessed, each going to a different set), change in associativity doesn't affect hit ratio.

In testcase #10 in testcases.txt
Cache size = 32 bytes, block size = 4 bytes, T = 4

Associativity	Hit ratio
1	0
2	0.262
4	0.459
8	0.852

In this case, all accesses are to the same set, so increase in associativity increases the hit ratio.

3) **Effect of T**

In inp_qsort.txt
Cache size = 64 bytes, Block size = 8 bytes, associativity = 2

T	Hit ratio
1	0.629
2	0.629
4	0.629
5	0.627
10	0.611
20	0.589
50	0.547

100	0.546

Thus small values of T give similar results, while excessively large values decrease hit ratio as blocks occupy cache space much longer after their last access.

In testcase #11 in testcases.txt
Cache size = 32 bytes, block size = 8 bytes, associativity = 2

T	Hit Ratio
1	0.885
2	0.846
3	0.808
4	0.769
6	0.692
8	0.615
16	0.308
32	0.038

Here an increase in T decreases the hit ratio. Here one element is accessed twice and then never accessed again, occupying space.

In testcase #12 in testcases.txt
Cache size = 32 bytes, block size = 4 bytes, associativity = 2

T	Hit Ratio
1	0.003
2	0.016
4	0.042
6	0.069
8	0.095
10	0.122

12	0.135

Here, one element is accessed at some intervals, while other elements are accessed just once. Larger T allows cache to retain that one element longer, thus increasing hit ratio.

Thus, increasing T can increase or decrease hit ratio, depending on the instructions.