CS252: Cache Simulator (15 points)

Due: Thursday, May 12

You are to write a C++ program that will act as a cache simulator

Cache basics:

As described in class, a cache consists of a number of Sets, each which contains a list of Blocks/cache lines. From a simulation point of view this allows flexibility in testing different configurations. One can have lots of Sets, each one with a single Block/cache line to effectively create a direct mapped cache. Or one can have a single Set holding all the Blocks, creating a fully associative cache. Or one can form any combination in the middle.

The memory address of the value read/written is used to determine which Set, Block, and Block offset within the Cache to which the memory location belongs. If $S$ is the number of Sets then you will need $\log_2(S)$ bits to represent the Set number. If the size of each Block (in bytes) is $B$, then the number of bits needed to represent the Block offset is $\log_2(B)$. Those amounts of bits are fixed, leaving the remaining bits in the address to represent the Tag which will be a unique identifier to determine which block is loaded into the cache line at a particular moment in time. This is much like a hash table where the Set Index is similar to a hash function to determine which location to place the item. And if multiple items map to the same location then the Tag is like the equals method to determine which particular item is stored at that location. We will be laying out our bits as: Tag, Set Index, Block Offset. So if you need 4 bits for the Set Index (i.e. you have 16 sets), and 3 bits for the Block Offset (i.e. you have 8 byte blocks), then for a 16 bit address, the first 9 bits will represent the Tag.

For reads, the cache should determine if the memory requested is stored in the cache. If it is then we get a cache hit. If it isn’t then the block with that memory location needs to be loaded from main memory into the cache. This is a cache miss. The address determines which Set it should be loaded into, but there may be multiple Blocks/Cache lines it could go in. If there is an empty one then it should go there. If they are all full then one needs to evict one of the existing blocks and replace it with the new block. The eviction policy we will implement is LRU – least recently used.

When a write happens there are several choices as well, but we will be implementing a write-back, write-allocate system. That is, if the memory is not in the cache then it should be loaded into the cache, just like with the reads. This is write-allocate. And when it is modified it should just change the cache value and not the main memory value. But it must remember it is now ‘dirty’ so that when it is evicted it can be written back to the main memory. This is the write-back part.
**C++ classes:**

We will be creating 4 classes for this project. Simulation, Cache, Set, and Block.

**Simulation** should just be an implementation file (.cc). It should contain just a main method (the only main method) and be used to drive the simulation. It should create a new Cache and then do a bunch of read/write tests to determine how the cache performs.

**Cache** is a C++ class and should be broken up into .h and .cc files. It should contain a constructor that takes in the size of the main memory address space (i.e. the size of a memory address in bits), the size of the cache in bytes, the size of the blocks in bytes, and the set associativity. Set associativity is the number of cache lines/blocks per set. Everything else you need should be able to be determined from those values.

The Cache class should also have methods to simulate a read and a write (those are good names for them!), which should take in a memory address (unsigned long works well for that). These methods should return a single int which will indicate how long the operation took: 0 for a cache hit, 1 for a cache miss, and 2 for a cache miss with a write-back on eviction.

There should also be a method to dump the cache which the program can call at the end. This should simply return how many cache values needed to be written back to main memory.

The Cache class should have a display method to dump the contents of the cache out to the screen for debugging.

Also in its implementation, your cache is going to need to hold onto a list of Sets. Simple arrays can certainly be used, but for this assignment I want you to use the vector class from the STL.

The **Set** class also will be split between an .h and .cc file. You should be able to construct a Set with a given number of blocks of a certain blocksize.

You will also want read and write methods as well as the cache class will defer a lot of the work to the Set class for these operations.

Just like in the Cache you will also need a display. You will also need something to hold onto the list of Blocks/cache lines stored in the set. Again, you are to use a vector for this purpose.

Lastly you will have a **Block** class which is also broken up into .h and .cc files. This is the class that holds a single block or cache line. You should be able to construct one with a given blocksize.
You will certainly need to hold onto the tag information for each block. You will also want something to indicate if the block is valid or not (i.e. has any information in it or not). Eventually you will want something to indicate if the block is dirty for the write-back. And you will also want some timestamp information for the LRU replacement algorithm. You can use the following for time information:

```c
#include <sys/time.h>

struct timeval tv;
struct timezone tz;
struct tm *tm;
gettimeofday(&tv, &tz);
gettimeofday(&tv, &tz);
tm=localtime(&tv.tv_sec);
hour = tm->tm_hour;
min = tm->tm_min;
sec = tm->tm_sec;
usec = tv.tv_usec;
```

However, you do not need to store the actual bytes that would actually go into the block when loaded. This is just a simulator for determining cache efficiency in terms of hits and misses and thus won’t need to store the actual data – just enough to know if there was a hit or miss.

**Compiling:**

You are to use the g++ compiler for this project.

**Testing:**

You will be given sample access runs (reads/writes) that you will have to code into your simulation driver. Between the displays of the cache and the numbers returned from the read/writes (0 – hit, 1 – miss, 2 – miss with writeback on eviction) you should be able to demonstrate that your cache is working correctly. Again, you will be given more information as the due date approaches on exactly which tests you should perform.

**Submission:**

You may work with a partner.

You are to submit a zip with your source code to the W: drive. You are also to turn in a cover sheet detailing your submission.