Problem 1 [60 points]: The L1 cache miss traces for six SPEC CPU 2006 applications are available in a zip file downloadable from the following URL.

https://www.cse.iitk.ac.in/users/mainakc/2020Autumn/traces.zip

Unzip the traces after downloading. It will create a directory named traces and place all the traces under that directory. You will see the following traces.

bzip2: two trace files (bzip2.log_l1misstrace_0, bzip2.log_l1misstrace_1)
gcc: two trace files (gcc.log_l1misstrace_0, gcc.log_l1misstrace_1)
gromace: one trace file (gromacs.log_l1misstrace_0)
h264ref: one trace file (h264ref.log_l1misstrace_0)
hmmer: one trace file (hmmer.log_l1misstrace_0)
sphinx3: two trace files (sphinx3.log_l1misstrace_0, sphinx3.log_l1misstrace_1)

These traces are binary files, so they won't open with any conventional editors. If you want to read about these applications, please visit http://www.spec.org/cpu2006/. The cases where you have two trace files, these are essentially two consecutive parts of the same trace. The applications with two trace files miss a lot in the L1 caches. As a result, the L1 cache miss trace is split into two parts to keep the size of each file within limit. You need to process these trace files in the order they are numbered.

Each trace entry has the following fields in that order.

```
char i_or_d;    // Instruction or data miss (you can ignore)

char type;      // A data miss or write permission miss
                // A zero value means write permission miss and you can
                // treat these as hits. A non-zero value means a genuine L1
                // cache miss.

unsigned long long addr;        // Miss address (64 bits)

unsigned pc;    // Program counter of load/store instruction that missed (32 bits)
```

For this problem, you won't require the i_or_d and pc fields.

Here is a code snippet that you can use to read the traces. Assume that you take the file name prefix (e.g., bzip2.log_l1misstrace) and the number of traces for the application under question (e.g., 2 for bzip2) from command line. These would be respectively argv[1] and argv[2].

```
int numtraces = atoi(argv[2]);
for (k=0; k<numtraces; k++) {
      sprintf(input_name, "%s_%d", argv[1], k);
      fp = fopen(input_name, "rb");
      assert(fp != NULL);

      while (!feof(fp)) {
         fread(&iord, sizeof(char), 1, fp);
         fread(&type, sizeof(char), 1, fp);
         fread(&addr, sizeof(unsigned long long), 1, fp);
         fread(&pc, sizeof(unsigned), 1, fp);

         // Process the entry
      }
      fclose(fp);
      printf("Done reading file %d!\n", k);
}
```

In this problem, you need to simulate a two-level cache hierarchy and pass the L1 cache miss trace through them. In the aforementioned code snippet, you need to call your two-level cache simulator where an entry is processed

and each entry will be passed through the cache simulator. I will call the two levels of the hierarchy L2 and L3. The L2 cache should be 512 KB in size, 8-way set associative, and have 64 bytes block size. The L3 cache should be 2 MB in size, 16-way set associative, and have 64 bytes block size. You need to report the number of L2 and L3 cache misses for each of the six applications for the following three cases. Assume that the L2 and L3 caches are empty before a trace is processed in each case.

A. L3 is inclusive of L2: An L3 miss fills into L3 and L2. An L3 eviction invalidates the corresponding block in L2.
B. L3 is non-inclusive-non-exclusive (NINE) of L2: An L3 miss fills into L3 and L2. An L3 eviction does not invalidate the corresponding block in L2.
C. L3 is exclusive of L2: An L3 miss fills into L2. An L2 eviction allocates the block in L3. An L3 hit invalidates the block in L3.

In all three cases, if an access misses in the L2 cache and hits in the L3 cache, the block must be copied from the L3 cache to the L2 cache requiring a replacement in the L2 cache. In the exclusive case, the copy of the block in the L3 cache must be removed at this time to maintain exclusion.

In inclusive and NINE caches, if an access misses in both L2 and L3 caches, you will need to imagine that the block is coming from memory and on its way back, it will first fill in L3 cache and then in L2 cache. So, you will invoke the replacement policy of the L3 cache first, pick an L3 cache victim, invalidate the L3 cache victim from L2 cache (needed only for inclusive), and finally invoke the replacement policy in L2 cache to pick an L2 cache victim.

In all cases, both L2 and L3 caches should execute LRU replacement. However, before invoking the replacement policy, you should look for an invalid way in the target set. The invalid ways should be filled first before replacing any valid block. Note that you only need to simulate the tags in the cache to measure the number of misses. The data values are not relevant. As a result, each level of cache is essentially a 2D array of tags where each row can be seen as a set. You also need to have the necessary LRU states per set.

When an access hits in the L2 cache, the LRU order of the block is updated only in the target L2 cache set. The LRU order of the block remains unchanged in the L3 cache. As a result, a block that is receiving a lot of hits in the L2 cache may become LRU in its L3 cache set and may get evicted. In an inclusive L3 cache, this will invalidate the block from the L2 cache. This is a big drawback of an inclusive cache hierarchy.

Prepare a table to summarize your results. Explain any difference in the number of misses seen across the three configurations. Only results without adequate explanation will attract sufficiently large penalty.

Problem 2 [40 points]: In the last problem, classify the L3 misses in the case of an inclusive L3 (case A) into cold, conflict, and capacity misses. Assume LRU replacement policy in the L3 cache. For each of the six applications, report the number of these misses in a table. For the fully associative implementation, try both LRU and Belady's optimal policies. Report the results separately for the two implementations. Offer adequate explanation for the results.

SUBMISSION:

Prepare a short write-up containing the results/explanations. The write-up must be typed (and not hand-written). We will accept only PDF submissions. All your results must be accompanied by appropriate explanation. Results without explanations may not receive any score. Include the source codes for the two problems in your submission along with a README file that explains how to run the codes and what the results generated by the codes mean. Please make sure that the

```
results printed by the codes are in a human-friendly format and
easily understandable. Mail your complete submission as a zip
ball to cs622autumn2020@gmail.com. Make sure to use only the
`zip' utility for preparing your submission. Avoid using any
other compression utilities due to possible portability issues.
Name your zip file as groupX.zip where X is your group id already
announced.

You are expected to submit your own work, which means you are not
allowed to discuss your work with any other group. Please refer to the
academic integrity policy of the course and the punishment if
you are caught cheating. You are free to discuss with me anything
related to the assignment. This is the safest way to get your
doubts clarified.
```