Project 2 – Due November 19
To be completed by the same groups as project 1, except when discussed with instructor

In this project, you will write a simplified simulator to evaluate distributed directory cache coherence protocols in CMPs. You will assume a tiled CMP with $P = 2^p$ tiles connected by a mesh NoC (network on a chip). Each tile includes a 32-bit CPU core with a private L1 cache (of size $N1 = 2^{n1}$ – in bytes) and an L2 cache (of size $N2 = 2^{n2}$). The L2 caches are shared among all cores. That is, the L2 nodules form a shared L2 cache of size P * N2 which is shared by all cores (NUCA). Each block in the L2 is associated with a directory entry to implement MSI coherence among the L1 caches. Assume that the block size in both L1 and L2 is $B = 2^b$ bytes and that the associativity of both L1 and L2 is $A= 2^a$. The distribution of the shared L2 entries into the individual L2 tiles is interleaved at the page level (4KB). That is, bits 12,…,12+p-1 of the 32-bits memory address (byte address with bit 0 being the lowest order bit) are used to specify the id of the L2 tile where this address is cached (the home tile). Note that the number of sets in each L2 cache tile is 2^m, where m = n2 – a (A-way associativity) - b (B bytes per block).

The simulator should assume a cross-bar interconnection between the tiles, with each tile having an outgoing message queue and an incoming message queue. In each cycle, at most one message is transmitted from the outgoing queue of each tile and at most one message can be received in the incoming queue of each tile. Two messages intended to the same destination should be sent in consecutive cycles.

An L1 hit is satisfied in the same cycle in which it is issued. The L2 hit time is *dc* cycles, and hence, the delay of a request that is not satisfied in L1 is equal to *dc* plus the delay caused by the messages needed for the coherence protocol. Finally, a miss in L2 will require sending a message to the on-chip memory controller, which is also connected to the cross-bar switch, a memory access, which is assumed to take *dm* cycles (uniform memory access), and a message back from the memory controller. Assume write-back caches with LRU replacement and a large write buffer between the memory controller and the memory, which takes write-backs from L2 out of the critical path (you do not have to simulate the write buffer – just ignore the time for write-backs from L2). A simple functional simulator of an associative cache written in C can be found in "cache.h" – you may use it as a starting point for your simulator. The TA will provide you a pointer to a cache simulator written in Java.

All the system parameters (p, n1, n2, b, a, *dc* and *dm*) should be specified in a configuration file, whose name is the first command line input to your simulator.

The simulator is trace-driven. That is memory load and store operations will be specified in an input trace file whose name is specified as the second command line input. Each line in the trace file is of the form:

cycle   core_id   R/W address

Indicating that at cycle "*cycle*", core "*core_id*" issues a read/write operation (*R/W* = 0 or 1, for read or write, respectively) to memory location "*address*" in Hex format. The entries in the trace file are sorted by *cycle* and then by *core_id*. The traces are generated assuming zero memory access delay. That is assuming that all memory requests are found in the L1 cache. In your simulation, you will insert the memory access delay due to L1 cache miss. That is, the time at which a request, R, is issued will depend on the memory delays of all the requests that are issued before R by the same core. In other words if the trace file indicates that two consecutive requests by the same core are issued in cycles c1 and c2, respectively, then the second request is issued c2-c1 cycles after the first request is satisfied.

Results reporting:

Your simulation should report the number of cycles needed for completing the execution of each core in the trace, the hit/miss rate in each cache module, the average miss penalty for each L1 miss in each core and the number of messages classified into control (short) and data (long) messages. You should also have a detailed debugging mode that can be invoked only when b=2 and small p, n1, n2 and short traces by arbitrarily specifying one last non-zero parameter at the command line. In that mode, a detailed log of messages is produced during execution and the content and state of each cache location is produced at the end of execution.

Test Traces.

Use the following as an example of a very short trace file. You should use this trace and other short traces to test and debug your simulator. This test file assumes 4 cores.

```
1   0 0 0x00000012
1   1 0 0x00000112
1   2 0 0x00000212
1   3 0 0x00000312
4   0 0 0x00000012
4   1 0 0x00000016
4   2 0 0x00000116
9   0 1 0x00000120
9   1 1 0x00000220
10 3 0 0x00000216
11 2 1 0x00000320
11 3 1 0x00000020
```

Comparative analysis:

After debugging your simulator, you should run your program on the two real traces found in /afs/cs.pitt.edu/courses/2410/Fall2019 with the following parameters:
P = 16 (p=4), n1= 13 (8KB L1 per core), n2=16 (64KB per L2 tile) , b = 5 (32 byte cache blocks), a = 2 (4-way associativity), *dc* = 4 cycles L2 cache access and *dm* = 20 cycle memory access latency.

After running with the above parameters, determine the effect of each of the following:

1) Doubling the size of the L1 cache,
2) Doubling the L2 cache size,
3) Doubling the associativity of the caches.