# Machine Problem 2

# Coherence Protocols

## Tasks:

### In this machine problem you will

In this project, individual students will implement a trace driven SMP simulator (shared multiprocessor simulator). You will be given a C++ generic-cache class, and you need to extend that class to work in a multi processor environment and to build various coherence protocols on top of it. You need to write it in C++ and you need to build it on a Linux machine. The most challenging part of this machine problem is to understand how caches and coherence protocols are implemented. Once you understand this, the rest of the assignment would be straightforward. The purpose of this project is to give you an idea of how parallel architecture designs are evaluated, and how to interpret performance data.

## Simulator

### How to build the simulator:

You will be provided with a working C++ class for a uni core system cache, namely `cache.cc, cache.h`. You can start from this point and build up your project by instantiating multiple cache objects to build your SMP system. Then you need to implement the required coherence protocols to keep all peer caches coherent. You have the choice not to use the given basic cache and to start from scratch (i.e. you can implement everything required for this project on your own). However, your results should match the posted validation runs exactly. You are also provided with a basic main function `main.cc` that reads an input trace and passes the memory transactions down through the memory hierarchy (in our case we have only one level in the hierarchy). Some of the code in `main.cc` is commented out and you need to fill in the correct code. A make file `Makefile` that generates the executable binary is also provided. In this project, you are supposed to implement one level of cache only, that is, you need to maintain coherence across one level of cache. For simplicity, you will assume that each processor has only one private L1 cache connected to the main memory directly through a shared bus. As shown the figure below:

Your job in this project is to instantiate these peer caches, and to maintain coherence across them by applying, MSI, MESI, and Dragon coherence protocols.

## Please note the following

* **It is recommended to always to go back to the book as a reference.**

  * (NEW) Fundamentals of Parallel Multi-core Architecture (Chapman & Hall/CRC Computational Science) 1st Edition-2015 by Yan Solihin (Author).

  * Fundamentals of Parallel Computer Architecture Paperback-2009 by Yan Solihin (Author).

* For now this repository is for FALL 2019 Class any future updates are the responsibility of future TAs if they want to use it.

* This framework developed on Ubuntu-Linux. It is your responsibility to make it work on windows or other systems, though it will run on any Linux machine that has a terminal.

## Problems

### Coding

For this problem, you are supposed to implement MSI, MESI, and Dragon coherence protocols and to match the validation runs given to you exactly. Your simulator should accept multiple arguments that specify different attributes of the multiprocessor system. One of these attributes is the coherence protocol being used. In other words, your simulator should be able to generate one binary that works with all coherence protocols. More description about the input arguments is provided in later section.

### Report

For this problem, you will experiment with various cache configurations and measure the cache performance with fixed number of processors. The cache configurations that you should try are:

* Cache size: vary from 256KB, 512KB, 1MB, 2MB while keeping the cache associativity at 8 and block size at 64B.

* Cache associativity: vary from 4-way, 8-way, and 16-way while keeping the cache size at 1MB and block size at 64B.

* Cache block size: vary from 64B, 128B, and 256B, while keeping the cache size at 1MB and cache associativity at 8 way.

* Cache policies: use write back and write allocate policy, and use LRU replacement policy for all cases. (both are already implemented in the provided code)

Do all the above experiments for MSI, MESI and Dragon protocols. For each simulation, run and collect the following statistics:

1. Number of read transactions the cache has received.

2. Number of read misses the cache has suffered.

3. Number of write transactions the cache has received.

4. Number of write misses the cache has suffered.

5. Total miss rate (rounded to 2 decimals, should use percentage format)

6. Number of dirty blocks written back to the main memory.

7. Number of cache-to-cache transfers (from the requestor cache perspective, i.e. how many lines this cache has received from other peer caches)

9. Number of interventions (refers to the total number of times that E/M transits to Shared states. Hint: There are two shared states in Dragon. See Chapter 8 for details)

10. Number of invalidations (any State->Invalid)

11. Number of flushes to the main memory.

12. Number of issued BusRdX transactions.

Present your statistics in both tabular format and figures as well. Finally, by comparing the behavior of various statistics, compare the performance with fixed number of processors and explain your observations. You should provide a detailed description with an insightful discussion to earn full credit.

## Getting Started

1. In order to “make” your simulator, you need to execute the following command: (You may need to make changes to the Makefile, if you add your own files.)

```
make clean; make; make clobber
```

2. After making successfully, it should print out the following:

```
---------------------------------------------------------

-----------FALL19-506 SMP SIMULATOR (SMP_CACHE)-----------

---------------------------------------------------------

Compilation Done ---> nothing else to make :)
```

3. An executable called “smp_cache” should be created.

4. In order to run your simulator, you need to execute the following command:

```
./smp_cache <cache_size>  <assoc> <block_size> <num_processors> <protocol> <trace_file>
```

#### Where:

* `smp_cache`: Executable of the SMP simulator generated after making.

* `cache_size`: Size of each cache in the system (all caches are of the same size)

* `assoc`: Associativity of each cache (all caches are of the same associativity)

* `block_size`: Block size of each cache line (all caches are of the same block size)

* `num_processors`: Number of processors in the system (represents how many caches should be instantiated)

* `protocol`: Coherence protocol to be used (0: MSI, 1:MESI, 2:Dragon)

* `trace_file`: The input file that has the multi threaded workload trace.

## Organization

* `02_FinalProject` - Machine Problem 2
  * `code` - base code for the problem
    * `trace`- trace data set
    * `val.v2` - validation data sets for the code
    * `src` - source code resides in this directory
      * `cache.cc`
      * `cache.h`
      * `main.cc`
      * `Makefile`

The directory `val.v2` contains validation runs you are asked to match, you have to literally match the output files, since we are going to use `diff` in order to spot out any differences. If your output does not match the given validations in terms of results and formats, you will be deducted points. You can use the following command to check if your output matches the given validation runs:

```
diff –iw <your_file> <validation_run>
```

Follow the same format specified in the validation runs.

The directory `trace` contains the trace files. Each trace file has a sequence of cache transactions, each transaction consists of three elements:

processor(0-7) operation(r,w) address(8 hexa chars) For example, if you read the line `5 w 0xabcd` from the trace file, that means processor 5 is writing to the address “0xabcd” in its local cache. You need to propagate this request down to cache 5, and cache 5 should take care of that request (maintaining coherence at the same time).

## Submission

You are supposed to submit all your source files, a Makefile and a report. You need to zip all of the mentioned files in one zipped folder named as `uintyID.zip` Using the following command:

```
zip unityID.zip *.cc *.h Makefile report.pdf
```

No folders are allowed in your zip file. We will test your project only on grendel `grendel.ece.ncsu.edu` machine, so make sure your project can be properly compiled (using make command) and run on grendel before submission.

## Grading Policy

Grade will be distributed as follows:

* 25%: MSI

* 25%: MESI

* 25%: Dragon

* 10%: Three secret runs (one for each protocol)

* 15%: Report

* If your output does not -exactly- match any validation run, you will be given only 10% for that part. (Instead of 25%)

* If your output does not match the secret runs, you will get 0% for that part.

* In order to get full credit for the report, you need to provide detailed discussion and performance evaluations results.

## Implementation suggestions

1. Read the given code carefully `cache.cc, cache.h`, and understand how a single cache works.

2. Most of the code given to you is well encapsulated, so you do not have to modify most of the existing functions. You may need to add more functions as deemed necessary.

3. In `cache.cc`, there is a function called `Cache::Access()` , this function represents the entry point to the cache module, you might need to call this function from the main, and to pass any required parameters to it.

4. You might create an array of caches, based on the number of processors used in the system.

5. In `cache.h`, you might need to define new functions, counters, or any protocol specific states and variables.

6. Start early and post your questions on the message board.