Fall'18

Cache Simulator

CS-572 COMPUTER ARCHITECTURE

FINAL PROJECT

Submitted By: **Akash Agarwal**
OSU ID Number: **933-471-097**

Introduction

This file describes the implementation details of Computer Architecture (CS-572) Final Project details that implements a **Cache Memory Simulator**. The executable accepts minimum two arguments, first one being the Cache Configuration file and the following arguments should be path to memory trace files that indicate memory access property. The Cache Memory Simulator is able to handle cache configurations with varying capacities (Number of indices), block sizes, levels of associativity, replacement policies (Random/ LRU), and write policies (Write-Through/ Write-Back).

After execution, the program generates one or more output files (one output file for each trace file) containing information on the number of cache misses, hits, evictions and the number of clock cycles pertained on each instruction execution.

Students were asked to take in account, the following assumptions:

- The simulator does not simulate on the actual data contents. We simply pretend that we copied data from main memory and keep track of the hypothetical time that would have elapsed.
- When a block is modified in a cache or in main memory, we always assume that the entire block is read or written.
- All memory accesses occur only within a single block at a time.

The Project is implemented in **C++ Language**, minimum required version: Cpp11

This Cache Simulator can be configured to run both as Single-Cache Configuration and Multi-Cache Configuration. The number of caches depend upon the number of trace files give as an argument to the executable. For example, one memory trace file will run the program in Single-Cache configuration.

Cache-Structure

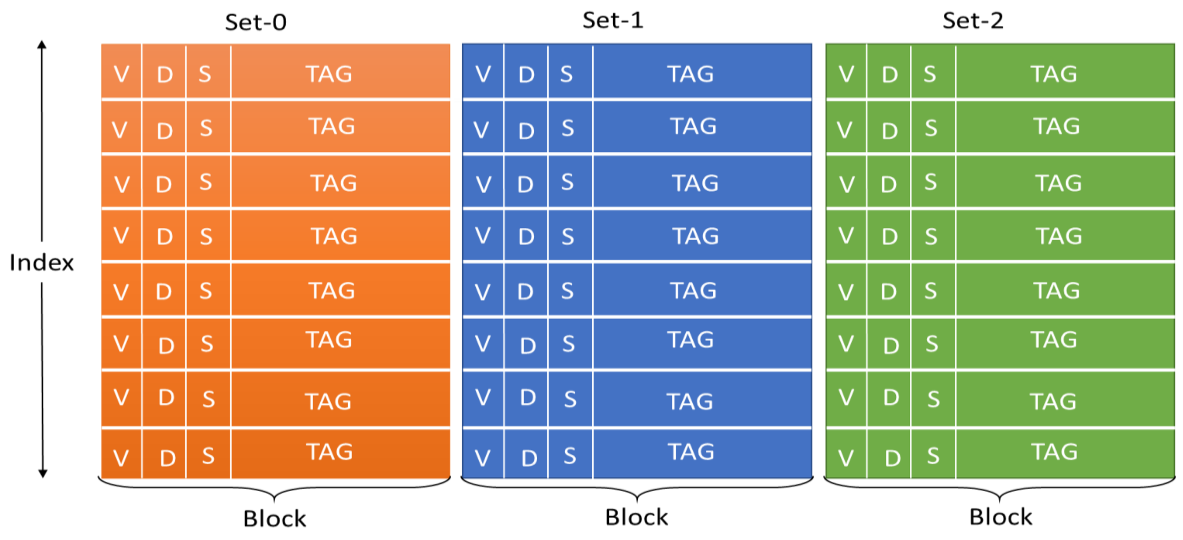


Figure-1: **Cache Representation**

Figure-1 shows the block-diagram of a 3-way associative cache with 8 cache entries.

In this implementation, every Cache consists of indices of cache entries. Each cache entry consists of 'n' cache blocks, where the value of 'n' depends upon the level of associativity mentioned in the configuration file. Each Cache Block consists of a **Valid Bit**, a **Dirty Bit**, a **Shared Bit** and **Tag Bits**.

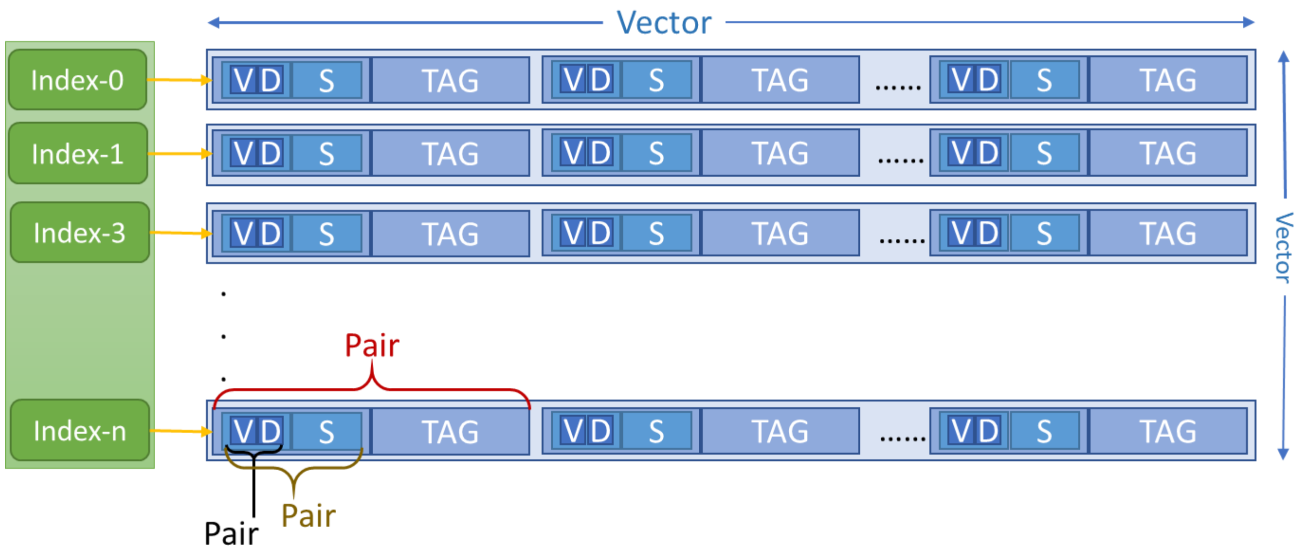


Figure-2: **Cache-Structure**

Figure-2 represents the combination of C++ Data Structures that have been used to form the structure of the cache, just as represented in Figure-1.

The following Data Structures have been used to construct Cache:

- std::pair
- std::vector

First, we define a std::pair of two boolean type variables, representing **Valid bit** and **Dirty Bit**.

```
typedef std::pair <bool, bool> nonDataBits; // nonDataBits.first represents Valid Bit
                                             // nonDataBits.second represents Dirty Bit
```

Next, define another std::pair of type 'nonDataBits' defined above and bool data type, representing **Valid bit**, **Dirty Bit** and **Shared Bit** respectively.

```
typedef std::pair <nonDataBits, bool> referenceBits;
                                   // referenceBits.first.first represents Valid Bit
                                   // referenceBits.first.second represents Dirty Bit
                                   // referenceBits.second represents Shared Bit
```

Next, define another std::pair of type ‘referenceBits’ defined above and unsigned long int type, representing **Valid Bit**, **Dirty Bit**, **Shared Bit** and **Tag Bits** respectively.

```
typedef std::pair <referenceBits, unsigned long int> cacheEntry;
                                    // cacheEntry.first.first.first represents Valid Bit
                                    // cacheEntry.first.first.second represents Dirty Bit
                                    // cacheEntry.first.second represents Shared Bit
                                    // cacheEntry.second represents Tag Bits
```

Now, each index of the cache should consist of these ‘cacheEntry’ elements. Also, each index will contain ‘n’ such cacheEntry variables (n-way Set Associative Cache). Thus, make a std::vector of cacheEntry type elements.

```
std::vector <cacheEntry> cacheSet(config.associativity);
// For config.associativity=3, create a vector, ‘cacheset’ containing 3 ‘cacheEntry’ blocks
```

The cache can consist of a number of indices. Thus, define a std::vector that can have std::vector<cacheEntry> as vector element, hence making a 2-D vector.

```
std::vector <std::vector <cacheEntry> > cache;
                                    // cache represents the structure of the cache
```

How to keep track of the number of clock cycles needed to execute memory access instructions?

It was given in the instructions that we have to assume, the number of CPU cycles it needs to access the cache = 13 and the memory access cycles = 230.
Every trace file consists of 3 types of memory instruction:

- Store Instruction (S): This is a write operation on memory address.
- Load Instruction (L): This is a read operation from a memory address.
- Modify Instruction (M): This instruction can be considered as a combination of Load and Store instructions. This instruction modifies a value at a memory location. It works by loading the data from the memory address and then storing back the changes made to the same memory location.

The program is keeping track of the Cache Hits and the Cache Misses and the Cache Writing Protocol (Write-Back or Write-Through). The code maintains a ‘**globalClockCycles**’ variable that keeps track of the total number of clock cycles incurred.

Every time the cache is accessed, **CacheAccessCycles** (= 13) are added to the **globalClockCycles** variable.
i.e. if an address gets a hit in the cache, only the time of accessing the cache will be added to the total cycles.
Whenever an address incurs a cache-miss, hypothetically, the cache will require to fetch the data of that address from the main memory, thus, in case of a cache-miss, an additional **MemoryAccessCycles** (= 230) time will be added to the **globalClockCycles** variable. Thus, for a miss, the total clock cycles incurred, will be 13 + 230 = 243 clock cycles.
The **globalClockCycles** value is also affected by the write-policy that the cache follows. The code can support two write-policies:

- Write-Through Cache: Every time there is a write on cache, update the memory. This will incur extra MemoryAccessCycles.
- Write-Back Cache: Every time there is a write on cache, update the value in the cache only, do not update the memory. The cache will incur extra MemoryAccessCycles, only when there is a cache-block eviction.

Thus, add the value of MemoryAccessCycles to the globalClockCycles variable while following the above stated write-policies.

Multi-Cache(Multi-threading Part)

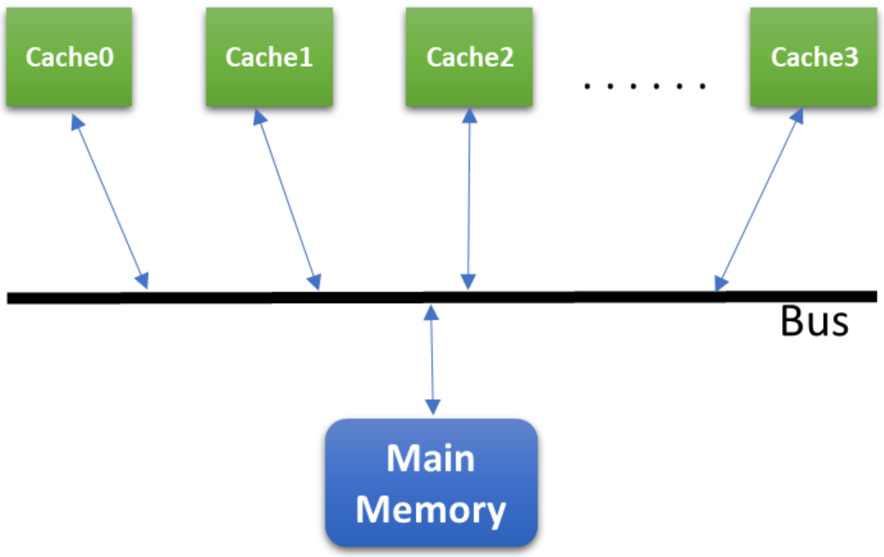


Figure-3: **Multi-Cache Structure**

As discussed above, the program can handle a Cache Configuration like the one given in the Figure-3, where multiple caches can work in parallel, such that they share a single memory bus, that connects them to the main memory.
Multiple- Caches running in parallel can run into the problem of accessing and manipulating the data of the same memory location simultaneously. Thus, making the implementation inconsistent. To cope up with the problem, the program implements the bus such that each cache should be

atomic access of the bus before requesting for the data. This will make the whole cache system consistent.
The following approach is being taken to implement **parallel cache simulator**:

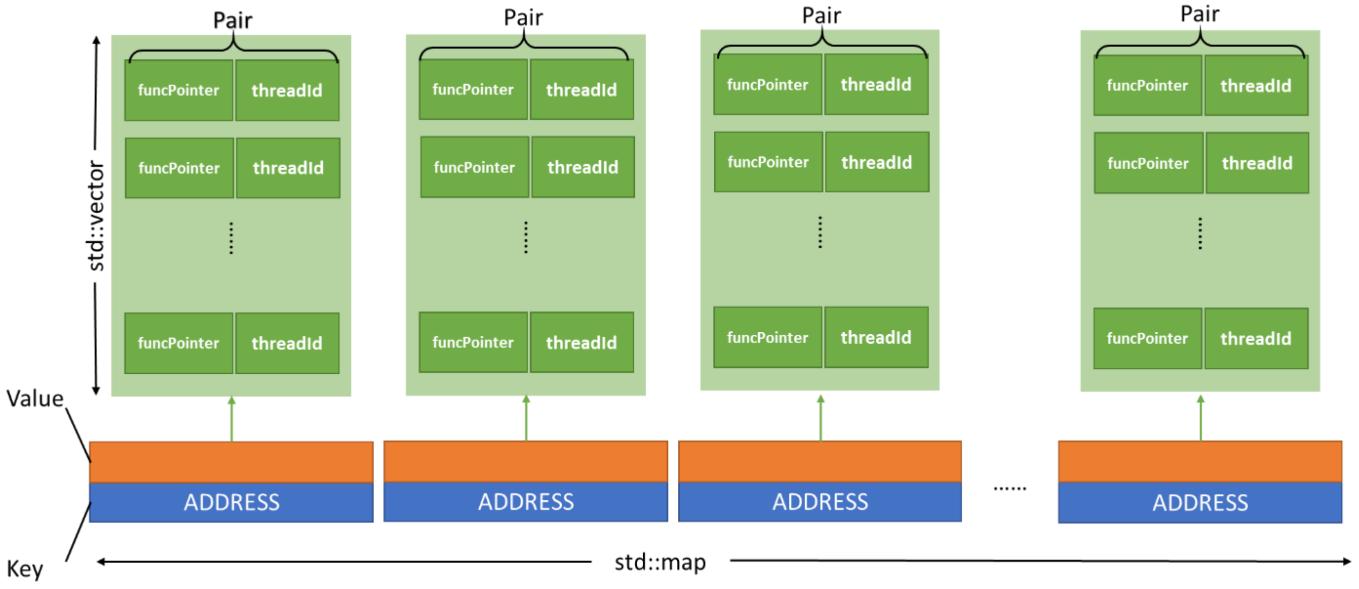


Figure-4: **Bus Structure**

Let's talk about the structure, then we'll look, its operation. The code makes use of following data structures to implement the Bus Structure:

- std::map
- std::vector
- std::pair

Apart from this, the bus implementation makes use of std::mutex and std::lock_guard() to provide atomic access to the bus.

Implementation

The cache_sim executable of the code, can take any number of arguments greater than 1(excluding itself). The number of trace files provided as arguments decide the number of simultaneously running caches to be made in the program. The code keeps track of the argc variable, and creates **argc – 2 threads**, for each trace file, where each thread represents the functioning of one independent cache.
Each thread will construct its own corresponding cache, using the method described before. To make the data coherent among all the caches, the code follows a simple '**write-invalidate cache bus snooping protocol**'.
Before initializing the threads, the main thread creates the bus object as shown in figure-4. The bus is a **std::map** where each key is an **address** of the memory location requested by the cache to the bus. Value of each key contains a **std::vector**. Each cache will have a **onBusResponse()** method that flips the values of shared bit in its entries or invalidates its entries. We'll define a **functionPointer** that points to the method. Each std::vector value will contain an element of type:

std::pair<functionPointer, threadID> where threadID is an int value and functionPointer is of type **std::function**.
Whenever a cache requires memory access, it will request the bus for memory access. The access to this common bus will be managed by **std::mutex** and **std::lock_guard()** (for atomic access).
For accessing the data from the bus, each cache will pass: address, message ('read', or 'write') and functionPointer to its onBusResponse() method.

The bus functions as follows:
For any request to the bus, the bus will keep track of the addresses that have been requested from it.

- If the requested address has never been requested by any cache before, the bus will create a new key (containing address). It will then create pair of the functionPointer provided by the thread and its threadID. It will create a vector, push that pair in that vector and add the vector as a value in that key.
- If the address has been requested before, i.e. it is able it find the address key, then simply make the pair of current thread's functionPointer and threadID and push it into the already existing vector, is the entry already does not exist.
- If the message = "read" and the address exists in the bus, for every element in the value vector call the onBusResponse() using the functionPointer with message as the parameter.
- If the message = "write" and the address already exists, for every element except the entry for current threadID, call the onBusResponse() using the functionPointer with the message as a parameter. Also, after calling onBusResponse(), erase the entry from the vector.
    - After doing the above task, check if the vector's size is 1, if true, call the onBusReponse() with the functionPointer and pass "notSharedAnymore" message as a parameter.

The onBusResponse() works as follows:

- If the message is "read" then at the address provided, change the shared bit to 1.
- If the message is "write" then at the address provided, invalidate the entry at that address.
- If the message is "notSharedAnymore" then at the address provided, change the shared bit to 0.

Thus, the act of bus calling the cache function (onBusResponse()) to invalidate the entries or updating the shared bit of the specific cache, is analogous to the snooping activity done by the caches in the actual bus snooping protocol.

Challenges Encountered

- The primary challenge was the implementation of the multi-cache system. The design of the bus and its processes, such that it co-relates with the actual cache coherence protocol required extensive thought process.
- Implementation of LRU (cache-block replacement policy). I was first implementing the cache structure such that each cache entry will contain a list of cacheSet elements, but the LRU policy implementation demanded taking the cacheBlock entry that is most recently accessed to be moved to the front of the list. List implementation was very inefficient in

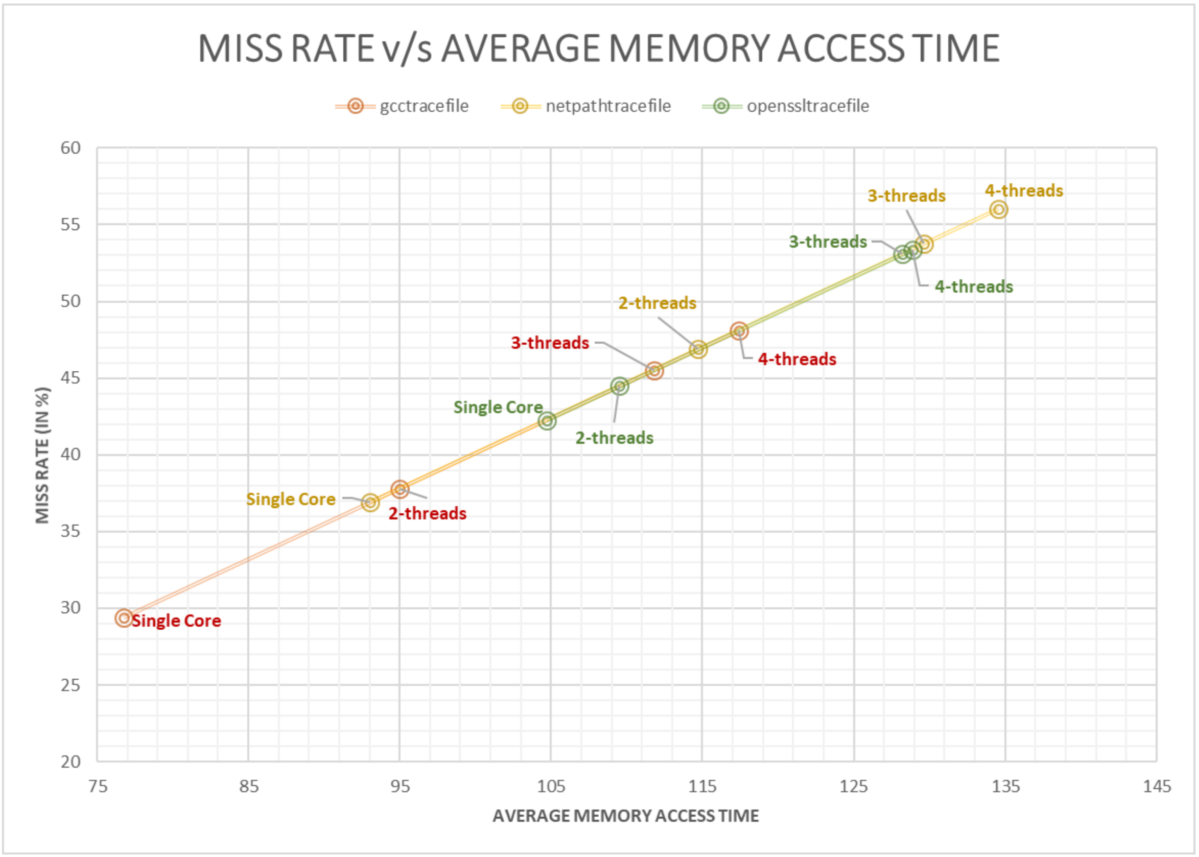
finding the index and moving the block. Therefore, I had to change the entire structure to vector of vector, instead of list of vectors.

- Finding out the corner cases for bus-snooping protocol and cache access method in cases of both write-back and write-through.

Things to implement differently if I were to re-implement the Project:

I would re-implement the bus. Due to shortage of time, it was not possible to choose the best data structure for the bus. The map as a bus, takes O(log(n)) time to search the address key in the bus. Also, the implementation of the function that provides the snooping functionality makes use of functions like std::find() which also takes O(log(n)) time. Also, the use of std::function, made the code very complex to understand and implement. So, I would think of some different approach if I re-implement the project. Also, for caches of large size, the cache structure will consume a lot of memory. This is because, the caches are implemented in std::vector and to make the execution speed of std::vector fast, C++ compiler dynamically allocates some extra memory for the future allocations in the vector. Thus, the actual memory acquired by the vector will be larger than the expected size.

How does the miss rate and average memory access time (in cycles) vary when the simulated program is operating in parallel?



The above graph shows the Miss rate versus Average memory Access Time graph for 3 different category of trace files provided by the instructor: gcctracefile, netpathtracefile and openssltracefile.

For each category, the executable is run for single core cache, 2-threads, 3-threads and for 4-threads. The value of miss rate and Average Memory Access Time (AMAT) were calculated from the output generated by all the threads, and then the average was taken to get a single value.
The values were then plotted on the graph above.
A general trend was noticed for all the categories of trace files.
It was observed that, the miss rate increase as we increase the number of simultaneously running caches in the system. The same trend was observed for the Average memory Access Time.
The above trend is expected. This is because, parallel threads (caches) race each other for the hold of the common bus. Also, for the data with the same address, their instructions can also have effect on the cache entries of the other caches (invalidation or updating of shared bits). The more the number of threads, the more will be their fight for mutex and also, there will be more invalidations. Thus, for a cache, the address which would have given a hit in uniprocessor cache, might give a miss in multi-processor implementation because its value would have been invalidated by some other parallel running cache.
Also, it was observed, that although there was a vast difference in Miss rate and AMAT in case of single processor cache and dual-processor cache, the difference was diminishing as were increasing the number of parallel running caches.

Extra Credit Task

- Implementation of Write-back cache:
  For the implementation of Write-back cache, the very first thing needed is the presence of a '**Dirty Bit**' in the cache entry. Thus, the cache structure described above having a Boolean data type representing the bit, was included.
  The Write-back policy will have impact on both the Load and Store instructions. Let's discuss each scenario in detail.

  Case-1: Store Instruction-
    - If Valid Bit = 0; Store the data in the cache block. Unlike Write-through policy, instead of updating the value in memory and incurring extra memoryAccessCycles, in case of Write-back, put the dirty bit = 1.
    - If tag matches and the valid bit is also 1; store the value at that address and make dirty bit = 1.
    - If the valid bit is not 0 and the tag also does not match, i.e. it is a case of eviction, if the dirty bit is 0, it will be a dirty eviction. Also, the block evicted needs to be written back at the memory. Thus, this step will incur extra memoryAccessCycles.

  Case-2: Load Instruction-
    - If Valid Bit = 0; No change, since no value is written to the cache, the only task will be to load the new value from the memory.
    - If Valid Bit = 1 and tag matches; it's a cache hit, no work needed
    - If neither the Valid Bit = 0 nor the tag matches; one of the cache blocks needs to be evicted. If the block to be evicted has Dirty Bit = 1, this will be dirty eviction. Also, the data needs to be written back to the memory, thus, it will incur extra memoryAccessCycles.

Since the Modify instruction makes use of Load and Store respectively, no additional task needs to be performed.

The above method of implementation was coded and the ‘cache_sim’ executable can be run with a write-back policy by changing its configuration with in the configuration file.