ECE4100/6100 CS4290/6290 - Fall 2024
Lab 4: CMP Memory System Design

Dr. Moinuddin Qureshi
Version: 1.0

Part ABC Due: November 17th 2024 @ 11:59 PM ET
Part DE Due: November 24th 2024 @ 11:59 PM ET
Part F Due: December 1st 2024 @ 11:59 PM ET

Changelog

- Version 1.0 (11/1/2024): Initial release.

1 Rules

- **This is an INDIVIDUAL assignment.** You may discuss this assignment with classmates, but you should code your assignment individually (i.e. no code sharing). **All honor code violations will be reported to the Dean of Students. NO EXCEPTIONS.**
- These lab assignments are very difficult and take a lot of time. It is in your best interest to start early.
- See the "Late Policy for Assignments" Canvas announcement for the course's late policy.
- Please utilize TA office hours, Piazza, and recitation if you have questions. **Do not go to the professor's office hours with lab-specific questions.**
- Read the entire document before starting. Not only is it critical to understanding the assignment, but most questions can be answered by reading the corresponding section.
- Sometimes, mistakes will be found in the PDF and the lab assignment. **It is solely your responsibility to ensure you are using the most up-to-date PDF and lab files.**
- Make sure that your code works with C++11 and on the provided reference machines as this what we will run the autograder on.

2 Introduction

As part of this lab assignment, you will build a multi-level cache simulator with DRAM based main memory. The system will then be extended to incorporate multiple cores, where each core has a private L1 (I and D) cache, and a shared L2 cache. Misses and writebacks from the shared L2 cache are serviced by a DRAM based main memory consisting of 16 banks and per-bank row buffers.

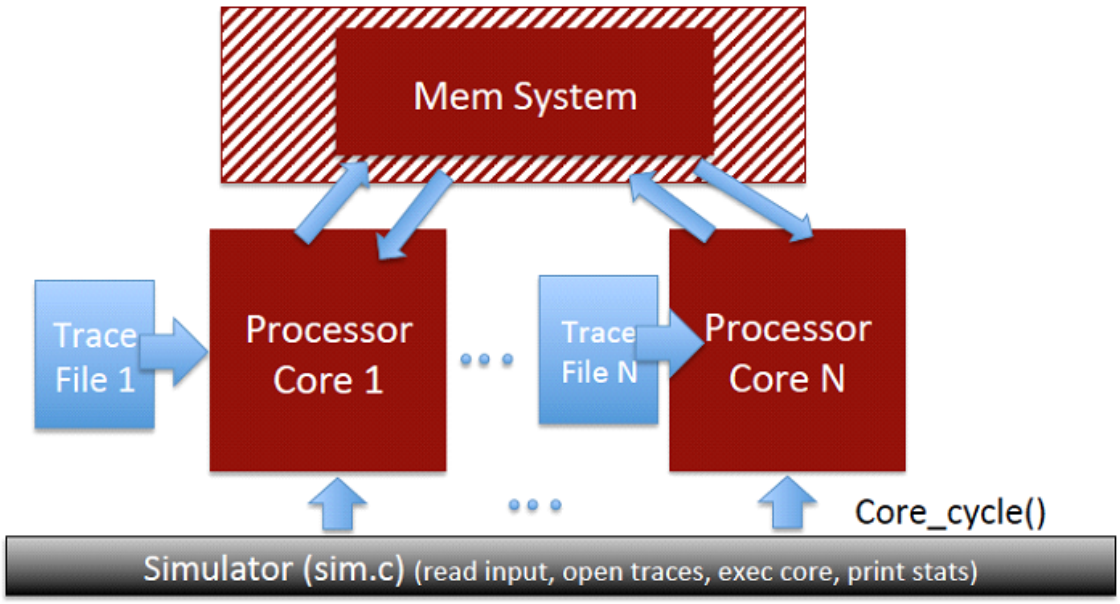

Figure 1: General overview of simulator

We will build this simulator in two phases. First Phase (A, B, and C) is for a single core system and the Second Phase (D, E, and F) extends the system to consist of multiple cores.

3 Design

3.1 Part A: Desgin a Standalone Cache Module (3 points)

In this part, you will build a cache model and estimate the cache miss rate. You need to implement your own data structure for the cache (named `Cache`) in `cache.cpp` and `cache.h` and leave all other files untouched (with the potential exception of `runall.sh`). The cache functions/methods must be written such that they will work for different associativity, line size, cache size, and for different replacement policies (such as LRU and Random). Look at Appendix A for more details on the cache implementation.

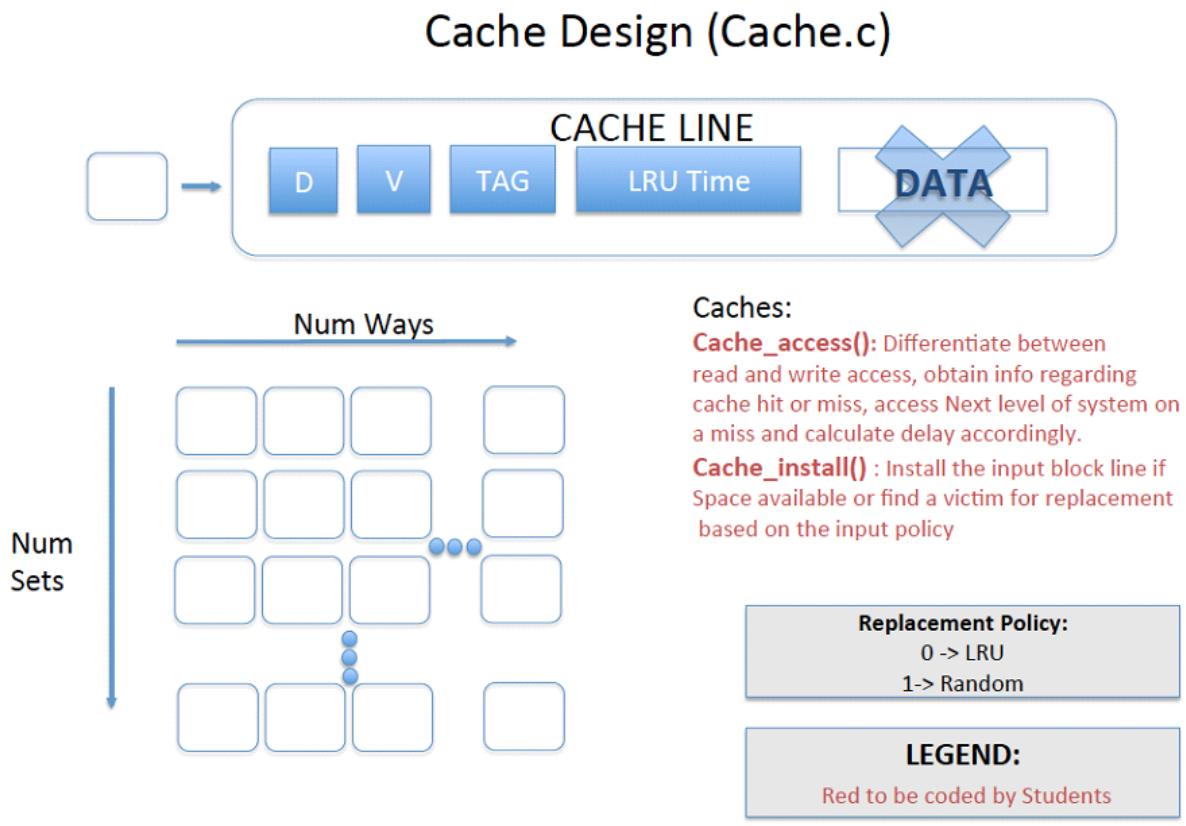

Figure 2: Overview of implementation for cache.cpp

As we are only interested in cache hit/miss information we will not be storing data values in our cache. We will provide you with traces for 100M instructions from three SPEC2006 benchmarks: `bzip2`, `lbm`, and `libq`.

Each trace record consists of 4 bytes of instruction PC, 1 byte of instruction type (0: ALU, 1: LOAD, 2: STORE) and 4 bytes of virtual address for LD/ST instructions.

We will provide you with the trace reader and the ability to change cache parameters from command line. The tracer calls the memory system for LD ST instructions and the function in the memory system in turn calls `cache_access()` for the DCACHE. If the line is not found in the DCACHE, the memsys function calls `cache_install()` for the DCACHE, which in turn calls `cache_find_victim()`. At the end, the memory system also calls `cache_print_stats()` for the DCACHE. The cache print stats functions are already written for you.

Your job is to write the following...

- `cache.h`
  - Fill in as needed based on your implementation.

- `cache.cpp`
  - `cache_new()`: Allocate and initialize your cache.
  - `cache_access()`: If the line is present in the cache if yes return HIT.
  - `cache_install()`: Install the line in the cache and track evicted line.
  - `cache_find_victim()`: Find the victim to be evicted.

The configuration used to run Part A is provided below.

```
./sim -mode 1 [trace]
```

3.2 Part B: Multi-level Cache (2 points)

You will implement an ICACHE, DCACHE, and connect it to a unified L2 cache and DRAM. You will also estimate timing for each request in this part.

Your job is to write the following...

- `dram.cpp`
  - `dram_new()`: Allocate and initialize your DRAM module.
  - `dram_access()`: Returns the delay incurred by a DRAM access and updates statistics.
- `memsys.cpp`
  - `memsys_access_modeBC()`: Returns the delay required to service a given request.
  - `memsys_l2_access()`: Called by `memsys_access_modeBC()` and returns the delay for servicing a request that accesses the L2 cache.

Note that for the purpose of calculating delay, you can assume that writebacks are done off the critical path and hence do not account for the accumulation of delay for a read request. Additionally, all caches are write-back and allocate-on-write-miss. For this part assume a fixed DRAM latency = 100 (provided by the constant `DELAY_SIM_MODE_B` in `dram.cpp`).

The configuration used to run Part B is provided below.

```
./sim -mode 2 -L2sizeKB 1024 [trace]
```

3.3 Part C: Implementing Simple DRAM (3 points)

For part C, you will model a simple row buffer for the DRAM memory system. You will assume that the DRAM memory has 16 banks and the memory system could follow either open page policy or close page policy.

Your job is to implement DRAM class in `dram.cpp` and `dram.h`:

- `dram_access_modeCDEF()` - Returns the DRAM delay for both open page and close page policy. This can be controlled by a command line argument. By default, it should take the fixed value as mentioned in the previous part.

Look at Appendix B for more details on DRAM implementation.

The configuration used to run Part C is provided below.

```
./sim -mode 3 -L2sizeKB 1024 -dram_policy [0/1] [trace]
```

3.4 Part D: Making the System Multicore (2 points)

You will conduct the effectiveness of your memory system for a multicore processor. In particular, you will implement a two-core processor and evaluate your design for three mix workloads: Mix1 (`libq-bzip2`), Mix2 (`libq-lbm`), and Mix3 (`bzip2-lbm`). You will model a simple LRU based replacement in the shared L2 cache and report the weighted speedup under LRU replacement. Pay attention to the memory traffic and the memory row buffer hit rate for the mix workload compared to the isolated workloads from part C.

Your job is to implement the following function in `memsys.cpp`:

- `memsys_access_modeDEF()` - Returns the delay required to service a given request.

The configuration used to run Part D is provided below.

```
./sim -mode 4 [trace]
```

3.5 Part E: Static Way Partitioning (2 points, extra credit)

Shared caches can cause a badly behaving application to completely consume the capacity of the shared cache, causing significant performance degradation to the neighboring application. This can be mitigated with way partitioning, whereby each core can be given a certain number of ways per set as their quota. In this part you will implement static way partitioning for a system consisting of two cores. We will provide `SWP_CORE0WAYS` as the quota for core 0 (the remaining `N − SWP_CORE0WAYS` becomes the quota of core 1).

Your job is to update the following function in `cache.cpp`:

- `cache_find_victim()` - Must now handle SWP.

The configuration used to run Part E is provided below.

```
./sim -mode 4 -L2repl 2 -SWP_core0ways [4/8/12] [trace]
```

3.6 Part F: Dynamic Way Partitioning (3 points, extra credit)

You need to implement any partitioning scheme different from Static Way Partitioning. One option is to implement a Utility-Based Cache Partitioning Scheme. You can read the paper, understand the scheme, and implement it whatever way you wish to. You are free to implement anything else by looking through other technical papers or even try something of your own.

You are allowed to add any structure you would like to implement the scheme. The only restrictions would be that you are not allowed to change the size, associativity, number of sets, or line size for any of the caches. All you have to do is to implement your choice of a non-static partitioning scheme which would decide how many ways of the L2 cache are dedicated to which core. Also note that no code debugging help would be provided for this part.

You will implement this scheme ensuring the code behaves as expected for the other modes of the Lab as well. Your scheme should be invoked by setting the command line parameters "-mode 4 -L2repl 3", i.e., as presented to you in the runall.sh file.

Write a report briefly describing the technique you have implemented, references, your idea on why it would provide better performance and the L2 miss percentages observed by running your code on the given traces.

The grading for this part will depend on your report, your understanding of the new scheme and your implementation.

+3 points if your implementation is correct and if you get a miss percentage lower than that obtained by Parts D and Part E (for SWP_CORE0WAYS = 50%) by more than 1% for any one of the three instruction mixes.

+1 point if your implementation is correct but you do not beat Parts D and E for any instruction mix.

4 Implementation Details

You have been provided the following files:

- `cache.cpp` - **(Part A & E & F)** Where your functions for providing cache functionality will go.
- `cache.h` - **(Part A)** Header file contianing structs and definitions necessary for your cache implementation.
- `memsys.cpp` - **(Part B & D)** Where your functions for providing memory access functionality will go.

- `memsys.h` - **Do not modify.** Header file containing structs and definitions necessary for `memsys.cpp`.
- `dram.cpp` - **(Part B & C)** Where your functions for providing DRAM functionality will go.
- `dram.h` - **(Part C)** - Header file containing structs and definitions necessary for implementing DRAM.
- `core.cpp` - **Do not modify.** Responsible for providing functionality for simulated CPU cores.
- `core.h` - **Do not modify.** Header file containing structs and definitions necessary for `core.cpp`.
- `sim.cpp` - **Do not modify.** Responsible for opening the trace, initialization, and instating and executing the simulator.
- `types.h` - **Do not modify.** Header file containing type declarations for enumerated types used throughout the code base.
- `Makefile` - **Do not modify.** A Makefile for compiling your code. Can be used to invoked the following commands...
  - `make` and `make all`: Compiles your code and creates an output file so you can run your code.
  - `make clean`: Cleans out compiled files.
  - `make fast`: Compiles your code with the `-O2` flag. See this for details.
  - `make debug`: Compiles your code with the preprocessor definition DEBUG defined. This causes code blocks of `#ifdef DEBUG ... #endif` to compile. **Highly recommend using for debugging purposes!**
  - `make profile`: Compiles your code for use with `gprof`. Helps diagnose bottlenecks if your code is running slowly.
  - `make validate`: Compiles your code and runs `runtests.sh`.
  - `make runall`: Compiles your code and runs `runall.sh`
  - `make submit`: Creates a tarball for your submission. Run `make runall` beforehand to generate `report.txt`. DO NOT change the name of the "src" directory, or your submission will not run with the autograder!
- `traces/` - **Do not modify.** A directory containing the execution traces for this lab.
- `scripts/` - **Do not modify.** A directory containing the following scripts...
  - `runall.sh`: Runs your code on the provided traces. The results of each run is placed in the correspondingly-titled `.res` file in `results/`. A summary of all the results can be found in `report.txt`, which is placed in whichever directory `runall.sh` is ran from (e.g. if you call `make runall` the report will be put in `src/`).

  - `runtests.sh`: Runs your code on a subset of the provided traces and verifies that your output statistics match the reference statistics for each test.

- `results/` - **Do not modify.** A directory containing the output of your code from `runall.sh`.
- `ref/` - **Do not modify.** A directory containing the reference outputs for a subset of the provided traces.
  - `results/`: A directory containing the reference outputs for a subset of the provided traces for each individual part of the lab.
  - `*.pdf`: Reports that summarize the reference outputs.

4.1 Simulator Parameters

The following command line parameters can be passed to the simulator:

- `-mode <num>`: Sets the mode to set the simulator to. There are 4 options:
  - `1`: Part A (default)
  - `2`: Part B
  - `3`: Part C
  - `4`: Part D/E/F
- `-linesize <num>`: Sets the cache line size in bytes for all caches (64 by default)
- `-repl <num>`: Sets the replacement policy for the L1 cache. There are 4 options:
  - `0`: LRU (default)
  - `1`: Random
  - `2`: SWP
  - `3`: DWP
- `-DsizeKB <num>`: Sets the capacity in KB of the L1 cache (32 by default)
- `-Dassoc <num>`: Sets the associativity of the L1 cache (8 by default)
- `-L2sizeKB <num>`: Sets the capacity in KB of the unified L2 cache (512 by default)
- `-L2repl <num>`: Sets the replacement policy for the unified L2 cache. There are 4 options:
  - `0`: LRU (default)
  - `1`: Random
  - `2`: SWP
  - `3`: DWP

- `-SWP_core0ways <num>`: Sets the static quote for core 0 in SWP (1 by default)
- `-dram_policy <num>`: Sets the DRAM page policy. There are 2 options:
  - `0`: Open-page (default)
  - `1`: Close-page
- `-h`: Print usage information

4.2 Simulator Statistics

The simulator keeps track of a variety of statistics, some of which you may have to update. Note that some statistics are indicated with an `X`, meaning that the statistic refers to a specific core. Depending on which part of the lab you are on, some statistics also have non-X variants, which simply indicate that it refers to the only core that the simulator used and primarily show up during the single-core portions of the lab. Additionally, the L2cache statistics do not have X variants (as the L2 cache is unified), but they are lumped in with the icache/dcache entries because they still specify the same information.

- `CYCLES`: Counts the number of simulated cycles it took for the simulator to run.
- `CORE_X_INST`: Counts the number of instructions that ran on core X.
- `CORE_X_CYCLES`: Counts the number of simulated cycles it took core X to run.
- `CORE_X_IPC`: The number of instructions completed per cycle by core X.
- `MEMSYS_IFETCH_ACCESS`: Counts the number of accesses to an L1 icache across all cores.
- `MEMSYS_LOAD_ACCESS`: Counts the number of load accesses to an L1 dcache across all cores.
- `MEMSYS_STORE_ACCESS`: Counts the number of stores accesses to an L1 dcache across all cores.
- `MEMSYS_IFETCH_AVGDELAY`: The calculated average delay of accessing the icache across all cores.
- `MEMSYS_LOAD_AVGDELAY`: The calculated average delay of performing a load to a dcache across all cores.
- `MEMSYS_STORE_AVGDELAY`: The calculated average delay of performing a store to a dcache across all cores.
- `(ICACHE/DCACHE/L2CACHE)_X_READ_ACCESS`: The number of read accesses to the icache/dcache/L2cache on core X.
- `(ICACHE/DCACHE/L2CACHE)_X_WRITE_ACCESS`: The number of write accesses to the icache/dcache/L2cache on core X.

- `(ICACHE/DCACHE/L2CACHE)_X_READ_MISS`: The number of read accesses to the icache/dcache/L2cache on core X that miss.
- `(ICACHE/DCACHE/L2CACHE)_X_WRITE_MISS`: The number of write accesses to the icache/dcache/L2cache on core X that miss.
- `(ICACHE/DCACHE/L2CACHE)_X_READ_MISS_PERC`: The percentage of read accesses to the icache/dcache/L2cache on core X that miss.
- `(ICACHE/DCACHE/L2CACHE)_X_WRITE_MISS_PERC`: The percentage of write accesses to the
  icache/dcache/L2cache on core X that miss.
- `(ICACHE/DCACHE/L2CACHE)_X_DIRTY_EVICTS`: The number of dirty cache lines evicted from the
  icache/dcache/L2cache on core X.
- `DRAM_READ_ACCESS`: The number of read accesses to DRAM.
- `DRAM_WRITE_ACCESS`: The number of write accesses to DRAM.
- `DRAM_READ_DELAY_AVG`: The calculated average delay of performing a read from DRAM.
- `DRAM_WRITE_DELAY_AVG`: The calculated average delay of performing a write to DRAM.

5 Debugging & Debug Outs

For debugging purposes, the Makefile has an option called `make debug`. This allows code enclosed in `#ifdef DEBUG ... #endif` blocks to compile alongisde your other code. For instance, say I have the following...

```
int x = 2;
#ifdef DEBUG
    x = 4;
#endif
printf("%d", x);
```

If you ran this with `make` by itself the output would be `2`. However, if you run it with `make debug` the output would be `4`. If we were to replace this example with `printf` statements, then we would only ever output when we call `make debug`!

Additionally, since debugging can be cumbersome we have provided debug outputs for you on Canvas. These debug outs are made by collecting the first X lines (indicated by the number on the debug outs folder) of the solution, then putting the output into a text file. In order to match these `debug_outs`, you **must** must add the `printf` statements provided in `debug_printfs.txt` to your code. The parameters for these statements have been purposefully left blank; you have to fill them in on your own depending on your implementation.

When adding the `printf`'s, be sure to wrap them in `#ifdef DEBUG` blocks so they only print when you call `make debug`.

To check your implementation using the debug outs, edit your code according to the type you're using. Then, run your code with the trace and the simulator parameters you want. Then, set your output to only go to X lines and set the destination of your output to some file. For example (with 100,000 lines):

```
./sim <necessary flags> ../traces/gcc.ptr.gz | head -n100000 > gcc.my_debug_out.res
```

After running the above, you can compare your output to the provided debug out's output using `diff`...

```
diff gcc.my_debug_out.res <provided gcc trace> > diff.txt
```

This will create a file `diff.txt` and place the output of the `diff` command inside it. At this point, you can go to the line numbers of your `.res` files indicated in `diff.txt` to see what is going wrong and fix it. If you run the `diff` command above and `diff.txt` is empty, you should call `make validate` to see if your results match!

6 Deliverables

To prepare your submission, run `make submit` to generate a tarball (`tar.gz`) with your code. **Do NOT change the name of the "src" directory**, as the autograder will not be able to test your submission. Make sure to untar your submission to ensure all the required files are there before submitting. **You are solely responsible for what you submit.** If you submit improperly, you can face up to a 10% grade deduction. Please do not make us use this rule and submit properly.

To see if you can receive full credit, you should run `make validate` and ensure you are passing. Note that just because you pass `make validate`, this does **NOT** mean you get a 100%. We reserve the right to examine any submission if we suspect plagiarism or attempts to game the autograder in any way.

7 Grading

This lab is worth 10 points in total. Note that this project is 10% of your final grade, meaning that each point equates to one point towards your final grade. There are 5 extra points of extra credit available by doing part E and F, meaning you can get 5 bonus points towards your final grade.

While there is no strict efficiency requirement, please ensure your simulator finishes each trace in less than a minute. This is to ensure expedient grading, since verification would be difficult and tedious otherwise. If your simulator is taking too long, we recommend you profile your code to see where most your code spends its time running. You can use `make profile` to allow your code to be used with `gprof`.

7.1 Grade Calculation

There are 6 parts of this project, each with their own number of points. The breakdown for points per part is provided below.

- **Part A**: 3 points
- **Part B**: 2 points
- **Part C**: 3 points
- **Part D**: 2 points
- **Part E**: 2 points, extra credit
- **Part F**: 3 points, extra credit

For each part of the project, your submission will be checked on a variety of simulator configurations. For each configuration, you will be evaluated based on how your simulator's metrics compare to reference metrics for `bzip2`, `lbm`, and `libq`. The metrics you are compared to are provided below.

- `CORE_0`: IPC only.
- `MEMSYS`: Only the `AVGDELAY` metrics.
- `ICACHE_X & DCACHE_X & L2CACHE`: All in output except the `PERC` metrics.
- `DRAM`: All `DRAM` metrics.

To receive full credit, you must match the reference metrics for each configuration of each test case. Partial credit is awarded for each metric depending on how close you are to the reference metric. The calculation for partial credit is provided below.

- $\pm$ **1%**: Full-credit (100%)
- $\pm$ **5%**: Half-credit (50%)
- $\pm$ **50%**: Quarter-credit (25%)

8 Reference Machines

The autograder will run on a reference machine to ensure that everyone conforms to one standard. There are two reference machines you can use for the course:

- **ECE**: ece-linlabsrv01.ece.gatech.edu (email)
- **CS**: shuttle3.cc.gatech.edu (help form)

How to access these machines depends on whether or not your are using Georgia Tech Wi-Fi (`eduroam`). If you are not, then you need to install the Georgia Tech VPN client (see Remote Access → VPN or Virtual Private Networking). Once you are connected either via VPN or Wi-Fi, you should be able to `ssh` into one of the above servers via the command line of your operating system like so:

```
ssh gburdell3@ece-linlabsrv01.ece.gatech.edu
```

Please note that `gburdell3` should be replaced with *your* GT username. If you receive `Permission denied` or `Access denied` when trying to connect, then use the above links to request access. Please include your GT username, the course name (ECE 4100/6100), and the instructor in the request.

We highly recommend that both CS and ECE students obtain access to each other's servers as these servers have been shown to go down at times. You can contact the respective help desks to request access. Since the servers can go down, it is crucial that you do not exclusively work on the servers while working on these projects. We recommend working on your code locally using services like WSL that will simulate a Linux environment for you to work and verify your implementation in.

9 FAQ

1. **How should I get started?**
   We strongly recommend that you read through the header files first to get a sense of what data structures are available to you and what you must implement. The header files should provide documentation for everything you need to know to complete this lab.

2. **Any helpful tips for the project?**
   Here's a couple:

   - You can use the timestamp method to implement LRU replacement. We have provided current_cycle as a means for tracking time.
   - You will need the last_evicted line for Part B, where you will need to schedule a writeback from the Dcache to the L2cache, and from the L2cache to DRAM.
   - If you pass one part of the project, there is no guarantee that your implementation will work correctly in later parts. For debugging, be sure to debug the current methods you made as well as the ones you made previously.
   - For `RANDOM` replacement, you may get minor changes in the miss rate compared to what is shown in the report.
   - You must follow the submission file names to receive full credit. However, you will not be penalized if Canvas automatically adds a numeric suffix to the filenames.

3. **I get an error when I try to execute `runall.sh` or `runtests.sh`**
   Check that you have execute permissions on both scripts. Add execute permissions to these files using this command: `chmod +x runall.sh runtests.sh`

4. **What do I do if I get a `no space left on device` error?**
   You can run `du -sh *` to see what is taking up space in your current directory and use `rm` and `rm -r` to remove files and directories respectively. If you are running `./runall.sh`, ensure that you have cleared out the `results/` directory as the files here can be very large if you ran `./runall.sh` with your print statements.

Appendix A - Cache Model Implementation

In the "src" directory, you need to update two files:

- `cache.h`
- `cache.cpp`

Following Data Structures may be needed for completing the lab. You are free to use any name, any function (with any number of arguments) to implement a cache. You are free to deviate from these structural definitions as well.

- **"Cache Line"** structure (e.g., CacheLine) will have following fields:
  - **Valid**: denotes if the cache line is indeed present in the Cache
  - **Dirty**: denotes if the latest data value is present only in the local Cache copy
  - **Tag**: denotes the conventional higher order PC bits beyond index
  - **Core ID**: This shall be needed to identify the core to which a cache line (way) is assigned to in a multicore scenario (required for Part D, E, F)
  - **Last Access Time**: helps to keep track of individual last access times of the cache lines, which helps identify the LRU way
- **"Cache Set"** structure, (e.g., CacheSet) will have:
  - **CacheLine Struct** (replicated "# of Ways" times, as in an array/list)
- Overarching **"Cache"** structure should have:
  - CacheSet Struct (replicated "# of Sets" times, as in an array/list)
  - # of ways (maximum should be 16)
  - Replacement policy
  - # of sets
  - Last evicted line (CacheLine type) to be passed on the next higher cache hierarchy for an install if necessary

Status Variables (Mandatory variables required for generating the desired final reports as necessary. Aimed at complementing your understanding of the underlying concepts):

- **stat_read_access**: Number of read (lookup accesses do not count as READ accesses) accesses made to the cache

- **stat_write_access**: Number of write accesses made to the cache
- **stat_read_miss**: Number of READ requests that lead to a MISS at the respective caches
- **stat_write_miss**: Number of WRITE requests that lead to a MISS at the respective cache
- **stat_dirty_evicts**: Count of requests to evict DIRTY lines

Take a look at "types.h" to choose appropriate datatypes.

The following functions would be needed for caches:

- `Cache *cache_new(uint64_t size, uint64_t associativity, uint64_t line_size, ReplacementPolicy replacement_policy)`: Allocate memory to the data structures and initialize the required fields. (You might want to use calloc() for this)
- `CacheResult cache_access(Cache *c, uint64_t line_addr, bool is_write, unsigned int core_id)`: the system provides the cache with the line address. Returns HIT if access hits in the cache, MISS otherwise. Also if is_write is TRUE, then marks the resident line as dirty. Update appropriate stats.
- `void cache_install(Cache *c, uint64_t line_addr, bool is_write, unsigned int core_id)`: Installs the line: determine victim using replacement policy (LRU/RAND). Copies victim into last_evicted_line for tracking writebacks. Finds victim using cache_find_victim. Initializes the evicted entry. Initializes the victim entry.
- `unsigned int cache_find_victim (Cache *c, unsigned int set_index, unsigned int core_id)`: You may find it useful to split victim selection from install.
- `void cache_print_stats (Cache *c, const char *label)`: Implemented for you, DO NOT MODIFY.

The following variables from `sim.cpp` might be useful (accessible via using `extern`):

- `SWP_CORE0_WAYS`
- `currect_cycle`

Appendix B - DRAM Model

In the "src" directory, you need to update two files:

- `dram.h`
- `dram.cpp`

Following Data Structures may be needed for completing the lab. You are free to use any name, any function (with any number of arguments) to implement a cache. You are free to deviate from these structural definitions as well.

- **"Row Buffer"** Entry structure (e.g., RowbufEntry) can have following entries:
  - **Valid**
  - **Row ID**: If the entry is valid, which row
- **"DRAM"** structure can have the following fields:
  - Array of Row Buffer Entry

Status Variables (Mandatory variables required for generating the desired final reports as necessary. Aimed at complementing your understanding of the underlying concepts):

- **stat_read_access**: Number of read (lookup accesses do not count as READ accesses) accesses made to the cache
- **stat_write_access**: Number of write accesses made to the cache
- **stat_read_delay**: Keeps track of cumulative DRAM read latency for subsequent incoming READ requests to DRAM (only the latency paid at DRAM module)
- **stat_write_delay**: Keeps track of cumulative DRAM write latency for subsequent incoming WRITE requests to DRAM (only the latency paid at DRAM module)

The following functions would be needed for caches:

- `DRAM *dram_new()`: Allocates memory to the data structures and initialize the required fields. (You might want to use calloc() for this)
- `uint64_t dram_access(DRAM *dram, uint64_t line_addr, bool is_dram_write)`: You may update the statistics here, and also call `dram_access_mode_CDEF()`
- `uint64_t dram_access_mode_CDEF(DRAM *dram, uint64_t line_addr, bool is_dram_write)`: You might need it for Part C and later. Assumes a mapping with consecutive lines in the same row. Assumes a mapping with consecutive rowbufs in consecutive rows. You may use this function to track open rows. You may compute delay based on row hit/miss/empty
- `void dram_print_stats(DRAM *dram);`: Implemented for you, DO NOT MODIFY.

DRAM Latencies

- `ACT`: 45
- `CAS`: 45
- `PRE`: 45
- `BUS`: 10

The following variables from `sim.cpp` might be useful (accessible via using `extern`):

- `SIM_MODE`
- `CACHE_LINESIZE`
- `DRAM_PAGE_POLICY`

Appendix - Plagiarism

Preamble: The goal of all assignments in this course is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives an unfair advantage over students who do the work and follow the rules.

1. As a Georgia Tech student, you have read and agreed to the Georgia Tech Honor Code. The Honor Code defines Academic Misconduct as "any act that does or could improperly distort Student grades or other Student academic records."

2. You must submit an assignment or project as your own work. Absolutely no collaboration on answers is permitted. Absolutely no code or answers may be copied from others — such copying is Academic Misconduct. NOTE: Debugging someone else's code is (inappropriate) collaboration.

3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "*submission of material that is wholly or substantially identical to that created or published by another person*").

4. Publishing your assignments on public repositories accessible to other students is unauthorized collaboration and thus Academic Misconduct.

5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.

6. Students suspected of Academic Misconduct are informed at the end of the semester. Suspects receive an Incomplete final grade until the issue is resolved.

7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.

8. If you are not sure about any aspect of this policy, please ask Dr. Qureshi