CS {4/6}290 & ECE {4/6}100 - Spring 2026
Project 1: Cache Simulator
Version 1.1

Professor Tom Conte

C1 Due: February 9th, 2026 @ 11:59 PM
C2 Due: February 20th, 2026 @ 11:59 PM

1 Changelog

- **Version 1.1**: 2026-02-07: Clarified incorrect hybrid prefetcher specification
- **Version 1.0.2**: 2026-02-07: Fixed minor typo in Appendix C
- **Version 1.0.1**: 2026-02-03: Clarified number of entries per Markov row (it's 4)
- **Version 1.0**: 2026-02-02: Initial release

2 Rules

- **This is an individual assignment. ABSOLUTELY NO COLLABORATION IS PERMITTED.** All cases of honor code violations will be reported to the Dean of Students. See Appendix A for more details.
- The due date at the top of the assignment is final. Late assignments will not be accepted.
- Please use office hours for getting your questions answered. If you are unable to make it to office hours, please email the TAs.
- This is a tough assignment that requires a good understanding of concepts before you can start writing code. **Make sure to start early.**
- Read the entire document before starting. Critical pieces of information and hints might be provided along the way.
- Unfortunately, experience has shown that there is a high chance that errors in the project description will be found and corrected after its release. **It is your responsibility to check for updates on Canvas, and download updates if any.**
- Make sure that all your code is written according to **C99 or C++11** standards, using only the standard libraries.

3 Overview

The goal of this project is to build a cache simulator ***and*** verify it is correct via the validation outputs of our reference simulator (validation outputs provided). Note that in this cache simulator, we are only simulating address/tag accesses; we will not be concerning ourselves with the actual data at each location.

You will implement the cache simulator in 2 different checkpoints. For checkpoint 1, you will implement a simple L1 cache. For checkpoint 2, you will extend the simulator to incorporate an L2 cache, a prefetcher, and then submit an experiments report. Each cache will be implemented with different eviction, insertion, and write policies, which are specified in later sections.

Please thoroughly read section 6 for specific details about the cache simulator.

Only For CS 6290/ECE 6100 Students - Please note

- You will implement two additional prefetcher schemes: a Markov prefetcher and a hybrid prefetcher.
- Add your observations of the above two in the report.

4 Background

This section describes various course topics that are important for this project.

4.1 Replacement Policies

In this project, we will consider 2 different cache replacement policies.

LRU eviction with MRU insertion (MIP): This is the standard insertion policy used with LRU replacement. The LRU block is evicted, and new blocks are inserted in the MRU position.

LRU eviction with LRU insertion (LIP): This insertion policy is designed for L2 caches to avoid thrashing on workloads with low temporal locality. Here, the LRU block is evicted, and new blocks are inserted in the LRU position.

4.2 Prefetching

Compulsory cache misses are misses that occur due to a memory block being accessed for the first time. These are considered *compulsory*, since for a newly used block to be inserted into a cache, a miss needs to occur to pull the block from memory.

However, compulsory misses *can* be avoided by implementing a prefetcher. In prefetching, data is fetched before it is actually needed. If effective, prefetching can improve performance by reliably predicting new accesses, thus reducing compulsory misses.

For this project, we will consider up to 3 different prefetching algorithms, all to be applied to the L2 cache (meaning, any prefetched entries are added to the L2 cache).

In the following sections, we use "block address" to refer to the part of the address containing the tag and index.

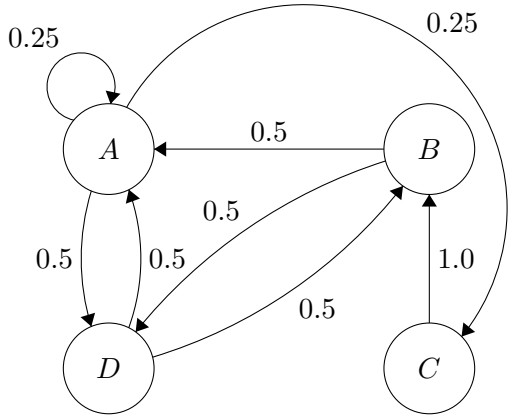
4.2.1 +1 Prefetcher

When an L2 read miss occurs on block address X, if the next block (the one at block address $X + 1$) is also not in the L1 or L2 caches, prefetch that block and insert it into the L2 cache.

4.2.2 Markov Prefetcher

NOTE: This is only for CS 6290/ECE 6100 students.

A Markov chain is a state machine where transitions from one state to another are random, defined by probabilities attached to the source state. Below is an example of a Markov chain:



In this Markov chain, starting from state A, there is a probability of 0.5 of going to state D, a probability of 0.25 of going to state C, and a probability of 0.25 of going back to state A.

The Markov prefetcher algorithm assumes cache misses occur as though they are states of a Markov chain. The transition between any two states X and Y represent the probability that a miss on block address Y occurs after a miss on block address X (based on the cache misses of the runtime of the program up to that point).

An example of a string of misses that could have created the Markov chain above is:

$$A\ D\ A\ D\ B\ A\ A\ C\ B\ D$$

In this string of misses, (A, D) appears twice, (A, A) appears once, and (A, C) appears once, implying that 50% of misses after A were block D, 25% were block A, and 25% were block C.

To represent a Markov chain in hardware, we create the following predictor table:

Block Address	Prediction 1	Prediction 2	Prediction 3	Prediction 4
D	$(A, 1)$	$(B, 1)$		
B	$(A, 1)$	$(D, 1)$		
C	$(B, 1)$			
A	$(D, 2)$	$(A, 1)$	$(C, 1)$	

The four columns of the table (the predicted next blocks) are ordered by most frequently used to least frequently used. The rows are ordered by most recently used to least recently used.

With the introduction of the Markov predictor, bound to L2, the following procedure is followed when an L2 cache miss occurs on block address B:

1. Look up the Markov predictor table to find the row with block address B.

   If row B exists, find the most frequently used block in the row, which we will call X. If there is a tie, pick the one with the largest block address (this is an arbitrary tiebreaker).

   Note: The reason we choose the most frequently used block, rather than probabilistically selecting is because the frequently used block is the most probable result to follow B in the Markov chain.

   Issue a prefetch for this block if all are true:

   - X is not present in L1 or L2
   - X is not the same as B

2. Update the Markov chain with the new miss. Let A be the previous block missed before B (if B is the first miss, this step can be skipped). Then,

   - **If row A exists and B is present as an entry**: Mark A as the most recently used row and increment B's frequency count in row A.
   - **If row A exists, but B is not present as an entry**: Mark A as the most recently used row, and insert $(B, 1)$. If there is no space to insert, then also evict the least frequently used entry (if there's a tie, arbitrarily evict the block with the lower address).
   - **If row A doesn't exist**: Insert a new row A at the most recently used position and add $(B, 1)$ as an entry to the row. If there is no space to insert, then evict the least recently used row.

Note that when entries are evicted from the Markov prediction table, they will not be written back anywhere; the evicted entry will be lost.

4.2.3 Hybrid Prefetcher

NOTE: This is only for CS 6290/ECE 6100 students.

This hybrid prefetcher combines both the +1 and Markov prefetchers to create a brand new prefetcher that is (hopefully) better than either two.

When an L2 read miss occurs on block address X, check the Markov table for entry X.

- **If no row entry is found for block address** X and the block address $(X + 1)$ does not exist in the L1 or L2 caches, prefetch block $(X + 1)$.
- **If a row entry is found for block address** X, prefetch as predicted by the Markov table.

In both cases, update the Markov table based on demand miss stream (i.e., follow step 2 of Markov).

5 Checkpoints

To make sure you're on the right track to completion, we have divided this project into two checkpoints:

- **Checkpoint 1 - Implement L1 Cache**: The L1 Cache has a write-back, write allocate (WBWA) write policy and uses MIP. Make sure to evaluate your results to match with the validation outputs.
- **Checkpoint 2 - Adding L2 Cache, Prefetcher(s), and Experimentation**: Implement an L2 cache with a write-through, write no allocate (WTWNA) write policy with LIP. Additionally, implement the required prefetching algorithms.

  Once your results are validated, you will need to create a report for the experimentation specified in Section 9.
  For CS 6290/ECE 6100 Students - Please note that your report will contain additional details as mentioned in Section 9.

6 Simulator Specifications

In this project, you will build a simulator for the cache hierarchy as outlined in the previous and following sections. Your simulator will read each trace of memory accesses and collect statistics that illustrate your cache's performance on each workload.

Here are some details about the cache simulator:

- The system uses 64-bit addresses.
- **The L1 cache** uses a write-back, write-allocate (WBWA) policy and uses MIP as its insertion policy.

  Since we are using a write-back policy, each block in the cache should include a dirty bit in its metadata to enable writebacks on eviction. Refer to section 6.2.2 for specific details about this operation.
- **The L2 cache** uses a write-through, write-no-allocate (WTWNA) policy and can either use LIP or MIP as its insertion policy.
- The L2 cache may also include a **prefetcher**, which can either be the +1 prefetcher (or Markov and Hybrid prefetchers for CS 6290/ECE 6100 students).

  You should also be tracking the prefetch hit and miss counts (the number of times a prefetched block is used versus immediately evicted). To facilitate this, each block should include a prefetch bit in its metadata. Refer to 6.2.5 for specific details about this operation.

6.1 Simulator Configuration

The L1 cache in your simulator will have the following configuration knobs:

- C_1: The size of the cache: 2^{C_1} bytes.
- B: The block size: 2^B bytes. This will also be the block size of the L2 cache.

  Restriction: The block size must be reasonable: $5 \leq B \leq 7$.

- S_1: 2^{S_1}-way associativity (i.e., each set contains 2^{S_1} blocks). When $S_1 = 0$, the cache is direct mapped. When $S_1 = C_1 - B$, the cache is fully associative.
- NOTE: L1 blocks have a dirty bit set on writes to a block. When a dirty block is evicted, a writeback should be performed to L2.

The L2 cache in your simulator will have the following configuration knobs:

- Enabled/Disabled
- C_2: The size of the cache: 2^{C_2} bytes

  Restriction: The size of the L2 cache must be strictly greater than the size of the L1 cache.
- B: Always equal to L1's B.
- S_2: 2^{S_2}-way associativity (i.e., each set contains 2^{S_2} blocks). When $S_2 = 0$, the cache is direct mapped. When $S_2 = C - B$, the cache is fully associative.

  Restriction: $S_2 \geq S_1$: The associativity of the L2 cache must be greater than or equal to the associativity of the L1 cache
- Replacement Policy: MIP, LIP
- Prefetcher: None, +1, Markov, Hybrid
- r: The number of rows in the Markov predictor table. For this project, we will keep the number of entries per row fixed at 4.

  Restriction: If the prefetcher is None or +1, then $p = 0$.
- NOTE: L2 uses the WTWNA write policy.

The tuple $(C_1, B, S_1, C_2, S_2, \{\text{MIP}, \text{LIP}\}, \{\text{None}, +1, \text{Markov}, \text{Hybrid}\}, r)$ fully specifies the cache configuration for this project.

6.2 Simulator Semantics

6.2.1 L1 And L2 Obliviousness

In this project, **the L1 cache does not know about the L2 cache, and vice versa**. The bus connecting L1 to L2 is the width of an L1 cache block. Effectively, the L1 cache believes it is talking to memory, and the L2 cache believes it is talking to the CPU. For example, the L2 cache sees a write-back from L1 as a write to L2. So even during writebacks from L1, your L2 cache simulation code should increment L2 hit/miss statistics counters.

Note that an L1 write miss triggers an L2 read to populate the L1 cache block, then performs a write on the L1 copy of the block. Additionally, when performing a write to L2, if the block is present in L2 already, that block should be moved to the MRU position. (If the written block is not present in L2, do not add it, because L2 is write-through and write-no-allocate.) The L2 cache is write-through, meaning that the values in the L2 cache are always in sync with values in memory.

6.2.2 L1 Writebacks to L2

Since the L1 cache uses a write-back policy, writes are not propagated to the L2 cache until the block is evicted. To enable this, you must maintain a dirty bit in each block's metadata, which tracks whether the block in the L1 cache has been written to since it was inserted into the cache. If the block is "dirty", you need to write back to the L2 cache when the block is evicted. Otherwise, if the block is "clean," you can evict the block without writing it back.

Thus, when the L1 cache misses, two things need to occur (in this order):

1. **Read from L2**: The L1 cache should access the missed block from the L2 cache.
2. **Evict and Writeback to L2**: The victim of the L1 cache should be evicted. If the block is dirty, the block needs to be written back to the L2 cache. Otherwise, the block can be removed from the L1 cache without writing back to L2.

In order to maintain consistency between solutions, these two operations **must** be performed in this order. Otherwise, if the block is written back and the requested block share an L2 set, their positions in the LRU queue for their L2 set could be swapped compared to the expected behavior.

6.2.3 Disabling Caches (checkpoint 1 only)

For simplicity, when the L2 cache is disabled, it should act as an zero-sized write-through cache: it should miss on every read, and send all writes directly to DRAM. L2 statistics still need to be updated and printed in this case. However, if the L2 cache is disabled, please set its hit time (HT) to zero.

6.2.4 On Inserting Prefetches

When an L2 read miss occurs, the L2 cache should **first** repair the miss and then a block should be prefetched (using the specified prefetching algorithm).

If the prefetcher produces a block address which is already in the L1 or L2 caches, the prefetched block should not be inserted into the L2 cache. Additionally, nothing should be done to the block (its LRU order of the block in the cache should not be updated as a result of the prefetch).

6.2.5 Prefetch Metadata Bit

Two statistics that the cache simulator should be tracking are prefetch hits and prefetch misses.

Prefetch hits are prefetched blocks which are accessed again (implying the prefetcher successfully predicted a requested block).

Prefetch misses are prefetched blocks that are evicted before they are ever accessed again (implying the prefetcher prefetched a block that did not need to be obtained).

To track this, a prefetch bit should be added to all blocks inserted into the L2 cache by the prefetcher. Then,

- When a block with the prefetch bit is accessed, the number of prefetch hits is incremented and the prefetch bit is cleared (this is to prevent a block from being marked as a hit multiple times).
- When a block with the prefetch bit is evicted, the number of prefetch misses is incremented.

6.3 Simulator Statistics

Your simulator will calculate and output the following statistics:

- Overall statistics:
  - Reads (`reads`): Total number of cache reads
  - Writes (`writes`): Total number of cache writes
- L1 statistics:
  - L1 accesses (`accesses_l1`): Number of times L1 cache was accessed
  - L1 hits (`hits_l1`): Total number of L1 hits.
  - L1 misses (`misses_l1`): Total number of L1 misses.
  - L1 hit ratio (`hit_ratio_l1`): Hit ratio for L1, calculated using the first three L1 stats above.
  - L1 miss ratio (`miss_ratio_l1`): Miss ratio for L1, calculated using the first three L1 stats above.
  - L1 AAT (`avg_access_time_l1`): See Section 6.3.1
  - Write-backs from L1 (`write_backs_l1`): Write-backs performed by the L1 cache; that is, the number of dirty blocks evicted from the L1 cache
- L2 statistics:
  - L2 reads (`reads_l2`): Number of times the L2 cache received a read request. This stat should not be touched unless there was an L1 read miss
  - L2 writes (`writes_l2`): Number of times the L2 cache received a write request. This stat should not be touched unless there was a write-back from L1
  - L2 read hits (`read_hits_l2`): Total number of L2 read hits. This stat should not be touched unless there was an L1 miss
  - L2 read misses (`read_misses_l2`): Total number of L2 read misses. This stat should not be touched unless there was an L1 miss
  - L2 read hit ratio (`read_hit_ratio_l2`): Read hit ratio for L2, calculated using earlier L2 read stats
  - L2 read miss ratio (`read_miss_ratio_l2`): Read miss ratio for L2, calculated using earlier L2 read stats
  - L2 AAT (`avg_access_time_l2`): See Section 6.3.1
- Prefetching statistics:
  - Prefetches issued (`prefetches_issued_l2`): Number of prefetches issued as a result of an L2 miss. This stat should not be touched unless a new value is inserted into the L2 cache as a result of a prefetch.
  - Prefetch hit (`prefetch_hits_l2`): Number of prefetches that results in a cache hit. Note that a prefetched block should not increment this stat more than once.
  - Prefetch misses (`prefetch_misses_l2`): Number of prefetches that are evicted from the L2 cache before the data is used.

6.3.1 Average Access Time

For the purposes of average access time (AAT) calculation, we assume that:

- For L1, hit time is $2\,\text{ns} + (S_{\text{L1}} \times 0.2\,\text{ns})$.
- For L2, hit time is $8\,\text{ns} + (S_{\text{L2}} \times 0.8\,\text{ns})$.
- The time to access DRAM is determined by

  $$\text{DRAM\_TIME} = \text{DRAM\_AT} + (\text{DRAM\_AT\_PER\_WORD} \times \text{WORDS\_PER\_BLOCK})$$

  - DRAM_TIME is the DRAM access time.
  - DRAM_AT (64ns) is the default base miss penalty for accessing DRAM.
  - DRAM_AT_PER_WORD (2ns) is the access time per word fetched.
  - WORDS_PER_BLOCK is the number of data words per block (we assume 8 byte data words).

The provided code includes constants for these values.

When computing the AAT for the L2 cache, which is write-through and write-no-allocate, use the L2 read miss ratio (`read_miss_ratio_l2`) as the miss ratio in the AAT equation.

When computing the L1 AAT, use the following equation from the lecture notes:

$$\text{AAT}_{\text{L1}} = \text{HT}_{\text{L1}} + \text{MR}_{\text{L1}} \times \text{MP}_{\text{L1}}$$

where the hit time HT_{L1} is as calculated above and MR_{L1} is the traditional L1 miss ratio (`miss_ratio_l1`).

Hint: in this project, what is the miss penalty of the L1 cache (MP_{L1} in the equation above)?

7 Implementation Details

We provide a driver, `cachesim_driver.cpp`, which parses arguments, reads a trace from standard input, and invokes your `sim_access()` function in `cachesim.cpp` for each memory access in the trace. The driver takes a flag for each of the knobs described in Section 6.1; run `./cachesim -h` to see the list of options.

The provided `cachesim.cpp` template file also contains `sim_setup()` and `sim_finish()` functions you should complete for simulator setup and cleanup (including final stats calculations), respectively. By default, the provided `./run.sh` script invokes the `./cachesim` binary produced by linking `cachesim_driver.cpp` with `cachesim.cpp`.

You are discouraged from modifying the `Makefile`, `./run.sh`, `cachesim_driver.cpp`, or the header file `cachesim.hpp`, because it will make it much more difficult for TAs to help with debugging.

If you need to review C or C++ (or are interested in learning), please refer to Appendix C and Appendix D for some starting info about C and C++.

7.1 LRU Implementation Recommendations

LRU can be implemented with timestamps, but there is a nasty bug that can occur when evicting a dirty block from the L1 during the same access in which an entry is written to the

L2. In this case, unless the timestamp is incremented before the evicted block is inserted into the L2 cache, there can be a tie when finding MRU in L2. For the purposes of this project, ties are illegal. Additionally, there are edge cases with using timestamps in which you may end up needing negative timestamps to implement LIP, which makes debugging confusing.

Alternatively, we suggest using either a C++ `std::list` or a doubly-linked list to implement LRU. For example, the head can represent MRU and the tail can represent LRU. If a hit occurs, the element can be moved to the MRU position.

7.2 Docker Image

We have provided an Ubuntu 22.04 LTS Docker image for verifying that your code compiles and runs in the environment we expect — it is also useful if you do not have a Linux machine handy. To use it, install Docker (`https://docs.docker.com/get-docker/`) and extract the provided project tarball and run the `6290docker*` script in the project tarball corresponding to your OS. That should open an Ubuntu 22.04 `bash` shell where the project folder is mounted as a volume at the current directory in the container, allowing you to run whatever commands you want, such as `make`, `gdb`, `valgrind`, `./cachesim`, etc.

> **Note**: Using this Docker image is not required if you have a Linux system available and just want to make sure your code compiles on 22.04. We will set up a Gradescope autograder that automatically verifies we can successfully compile your submission.

8 Validation Requirements

We have provided six memory traces in the `traces/` directory of the provided project tarball. Validation outputs for the default simulator configuration for each of these traces are provided in the `ref_outs/` directory. There are also validation outputs for `gcc` under the default configuration except with only L1 (`-D`) and with the different prefetchers (`-F plus1`). Given these configurations, the output of your code must match these reference outputs byte-for-byte! We have included a script to compare the output of your simulator to these reference outputs using `make validate_undergrad` and `make validate_grad` depending on the section you belong in.

We would advise against modifying `cachesim_driver`, which prints the final statistics, unless you are confident it will not cause differences from the solution output. We may grade by testing against other configurations or other traces.

8.1 Debug Traces

Debugging this project can be difficult, particularly when looking only at final statistics, so we have also provided some verbose debug traces corresponding to the reference traces in a separate tarball on Canvas. Warning: it is large when decompressed, around 1.2 GB. Please see the `README.txt` in that tarball for more information. **You are not required to output or match these debug traces at all – they are only intended to be a debugging aid if needed**.

Note that your debugging statements should only print when a special `DEBUG` flag is set. Otherwise, it will slow down your implementation and potentially cause issues during grading. To make the debugging statements conditional, use the following code:

```
#ifdef DEBUG
    // Your debug statement here
#endif
```

To see your debugging statements print when you test your implementation, add `DEBUG=1` to the `make` command.

9 Experiments (Part of Checkpoint 2)

Once you have your simulator working, you should find the best cache configuration for each of the six traces that meets the following constraints:

1. Configuration should respect all the restrictions mentioned in Section 6.1.
2. The budget for L2 data store is 128 kB. Assume min. budget for L2 data store is 64 kB.
3. The budget for L1 data store is 32 kB. Assume min. budget for L1 data store is 16 kB.

The best cache configuration would have the lowest average access time, while minimizing data store and metadata/tag storage. Identify points of diminishing returns in terms of cache parameters (C, B, S). Consider that highly associative caches use substantially more area and power. Include any rationale, quantitatively or qualitatively, to justify your decisions.

In your report, in addition to the cache configuration and data store size, you must also provide the total size of any associated metadata required to build the cache for your recommended configuration(s). For all of the L1 and L2, the data storage size is 2^C **bytes**, respective to C for the associated cache. For L1, the per cache tag storage size is $2^{C-B} \times (64 - (C - S) + 2)$ **bits** due to the dirty and valid bits.

For L2, the tag storage size is $2^{C-B} \times (64 - (C - S) + 1)$ **bits**, since there are no dirty bits in L2. Assume that maintaining LRU information has no implementation cost for the simplicity of the project.

For CS 6290/ECE 6100 students please note
In your report, make sure to add:

- Experimentation on the additional prefetching algorithms (Markov and Hybrid). Assume p can be one of $\{0, 16, 32, 64, 128, 256, 512\}$.

You should submit a report in PDF format (inside your submission tarball) in which you describe your methodology, rationale, and results.
NOTE: to run your code faster, you can add `FAST=1` to the `make` command. This will enable -O2 optimization while compiling your code.

10 What to Submit to Gradescope

Please submit your project to Gradescope as a tarball containing the your experiments writeup as a PDF, as well as everything needed to build your project: the `Makefile`, the `run.sh` script, and all code files (including provided code). You can use the `make submit` target in the provided Makefile to generate your tarball for you. Please extract your submission tarball and check it before submitting!

We plan to enable a Gradescope autograder that will verify that your code compiles on 22.04 at submission time. The Gradescope autograder result is NOT indicative of your grade on the project and will only check that the assignment compiles.

11 Grading

By the checkpoint 1 deadline, the following criterion must be followed:

+10: Your simulator **completely matches** the validation output for checkpoint 1.

By the checkpoint 2 deadline, the remaining criteria must be followed:

+0: You don't turn in anything (significant) by the deadline

+50: You turn in well-commented significant code that compiles and runs but does **not** match the validation

+25: Your simulator **completely matches** the validation output for checkpoint 2

+10: You ran experiments and found the optimum configuration for the 'experiments' workload and presented sufficient evidence and reasoning.

Note: You can only run experiments after your simulator is completely validated.

+5: Your code is well-formatted, commented, and does not have any memory leaks! Check out the section on helpful tools.

Appendix A - Plagiarism

Academic Misconduct

Preamble: *The goal of all assignments in this class is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. Students who do the work and follow the rules should not be disadvantaged by those who do not.*

1. **As a Georgia Tech student, you have read and agreed to the Georgia Tech Academic Honor Code.** The Academic 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, AI assistants, 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 or persons*").
4. Publishing your assignments on public repositories, github, etc., that are 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 and powerful forensic techniques to determine whether there is copying of a coding assignment.
8. Submitting an assignment with code or text from an AI assistant (e.g., ChatGPT) **is academic misconduct (see below).**
9. **If you are not sure about any aspect of this policy, please ask Dr. Conte**.

Advice For Using AI Assistants

Anything you did not write in your assignment will be treated as an academic misconduct case. If you are unsure where the line is between collaborating with AI and copying AI, we recommend the following heuristics:

Heuristic 1: ***Never hit "Copy"*** within your conversation with an AI assistant. You can copy your own work *into* the conversation, but ***do not copy anything back*** from the conversation into your assignment. Instead, use your interaction with the AI assistant as a learning experience, and then let your assignment reflect your improved understanding.
Heuristic 2: ***Do not have your assignment and the AI agent open at the same time***. Similar to the above, use your conversation with the AI as a learning experience, then close the interaction down, open your assignment, and let your assignment reflect your revised knowledge. This heuristic ***includes*** avoiding using AI directly integrated into your composition environment: just as you should not let a classmate write content or code directly into your submission, so also you should avoid using tools that directly add content to your submission.
Note: Deviating from these heuristics does not automatically qualify as academic misconduct; however, ***following these heuristics essentially guarantees your collaboration will not cross the line into misconduct.***

Appendix B - Helpful Tools

You might find the following tools helpful:

- gdb: The GNU debugger will prove invaluable when you eventually run into that segfault. The Makefile provided to you enables the debug flag which generates the required symbol table for gdb by default.
  - To pass a trace on standard in (as cachesim expects) while running in gdb, you can invoke gdb with `gdb ./cachesim` and then run `run <cachesim args> < traces/mcf.trace` at the gdb prompt
- Valgrind: Valgrind is really useful for detecting memory leaks. Use the following command to track all leaks and errors:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
    ./cachesim <cachesim args> < traces/mcf.trace
```

Appendix C - Bitwise Operations in C

Bitwise operations will be very important to this project, as you will frequently need to apply bit masks to cache addresses to obtain the tag and index.

Here is a brief overview of bitwise operations in C:

- **Bitwise AND** (`val & mask`): This takes the bits of each operand and performs an AND on them. For example, `(0x57 & 0xBB)` becomes:

```
  01010111 (0x57)
& 10111011 (0xBB)
----------------
  00010011 (0x13)
```

  Bitwise AND is also used to "clear" bits (update bits to 0). If we treat the first operand as the "value" and the second as the "mask," then a bitwise AND passes all of the bits from the value, except for when the "mask" is 0.

```
  01010111
& 10111011
----------
  00010011
```

- **Bitwise OR** (`val | mask`): This takes the bits of each operand and performs an OR on them. For example, `(0x57 & 0x48)` becomes:

```
  01010111 (0x57)
| 01001000 (0x48)
----------------
  01011111 (0x5F)
```

  Bitwise OR is also used to "set" bits (update bits to 1). If we treat the first operand as the "value" and the second as the "mask," then a bitwise OR passes all of the bits from the value, except for when the "mask" is 1.

- **Bitwise NOT** (`~val`): This flips all of the bits of the operand. For example, `~0x57` becomes:

```
 ~ 01010111 (0x57)
-----------------
   10101000 (0xA8)
```

- **Bitwise XOR** (`val ^ mask`): This takes the bits of each operand and performs an XOR on them. Like bitwise AND and OR, this can be used for masking. If the mask is 1, then the corresponding bit from the value is flipped.

- **Left shift** (`val << shift`): This operation moves the bits of an integer right (towards the MSB) by the specified shift value, filling empty space with 0. For example, `(0x57 << 3)` becomes:

```
    01010111 (0x57)
<<         3
-----------------
    10111000 (0xB8)
```

  Notably, left shift is useful for making masks or rearranging blocks of bits.

  - `(1 << n)` will create a bitmask which is all 0s, except for a 1 in the nth position. For example, `1 << 2 == (0b00000100)`.
  - `(1 << n) - 1` will create a bitmask which is 1s between bits 0 and $n-1$. For example, `(1 << 4) - 1 == (0b00001111)`.
  - `(tag << 3)` will create a value which is the tag but moved 3 bits to the left (towards the MSB).

- **Logical/unsigned right shift** (`val >> shift`): This operation moves the bits of an integer right (towards the LSB) by the specified shift value, filling empty space with 0. For example, `(0x57 >> 3)` becomes:

```
    01010111 (0x57)
>>         3
-----------------
    00001010 (0x0A)
```

  Logical right shift is also useful for rearranging blocks of bits. For instance, if we know the index is at `addr[9:3]` (7 bits long), we can extract the index with `(addr >> 3) & ((1 << 7) - 1)`.

Appendix D - Guidelines for Using C++

This appendix serves as a very brief explainer for certain C++ concepts for those who have never used C++ before and want to use it for their project. This is not meant to be comprehensive; you may use any C++11 language features for your project!

Structs

On the most basic level, C++ syntax for structs closely aligns with C syntax for structs. For example, the following struct is valid in both C and C++:

```
struct Foo {
    int bar;
    int baz;
};
```

C++ does enable some special new features for structs, though! Structs are automatically `typedef`'d, meaning that you can use `Foo` (instead of `struct Foo`). Additionally, you can define methods in structs:

```
struct Foo {
    int bar;
    int baz;
    void do_really_cool_thing() {
        // you can reference bar and baz in here
    }
}
```

Methods can then be invoked with `foo.do_really_cool_thing()`. We encourage you to use helper methods to manage the complexity of your code!

Constructors and Initializers

You can define a constructor to override how a struct can be initialized. Be careful – fields will be initialized in the order in which they are declared:

```
struct Foo {
    // ...
    Foo(int arg): bar(arg), baz(arg * 2) {}
}
```

In this constructor definition, we use the name of the struct (`Foo`) and arguments to accept in parentheses. Then, we declare how struct fields are initialized from the arguments (yes, your eyes do not deceive you; the initializations are outside of the function body). You can also additional logic within the function body if additional work needs to be done.

You can then invoke the constructor is as follows:

```
Foo foo1(3); // initializes a Foo with bar=3, baz=6

Foo foo2; // this can only be used if there is a default constructor (i.e., Foo())
          // or no constructor (in which case, C++ creates a default constructor)
```

`new` and `delete`

Despite the name, `new` in C++ is not like `new` in Java and you generally shouldn't try to use it to "create a new object." Instead, use the syntax above (or something like `Foo(3)` without `new`).

`new` and `delete` are, in essence, type-safe variants of C's `malloc` and `free`. `new` creates space for the specific object on the heap and initializes it (using the specified constructor). `delete` destroys the object and frees the space. For example,

```
Foo *foo = new Foo(3); // allocates space for Foo on the heap,
                       // invokes constructor Foo(3);

delete foo; // destruct foo and free space on the heap
```

These operators can also be used to allocate heap arrays:

```
Foo *array = new Foo[3] { ... }; // allocates space for 3-element array
                                 // of Foo on the heap

delete[] array; // This special syntax should be used to free arrays.
```

You largely shouldn't use `new` or `delete` for your project as they force you to manually manage heap memory. Instead, you should use standard initializers (as described in the previous section) and use the standard library containers (e.g., `std::vector<T>`) that automatically manage their own heap allocations. This includes automatically freeing heap memory they had allocated by the time they go out-of-scope.

References

By default, when assigning a variable, data is copied. For instance,

```
Foo foo1 { /* imagine Foo is a really big struct */ };
Foo foo2 = foo1; // copy foo1 into foo2. quite inefficient...
```

This is also true for passing data into functions:

```
void process_data(Foo foo);
process_data(foo1); // copy foo1 into the function
```

To prevent this, you can use reference types. Items passed into a reference variable are aliased rather than copied:

```
Foo foo1 { ... };
Foo& foo2 = foo1; // no copy

foo2.bar = 3; // equivalent to foo1.bar = 3
```

You can also use references with functions:

```
void process_data(Foo& foo); // pass "foo" in by reference, avoiding a copy
float summarize_foo(const Foo& foo); // const references will avoid a copy and will
                                     // prevent foo from being mutated within the function
Foo& acquire_foo(); // do something and then return a reference to a Foo
```

STL Containers

The standard library provides a bunch of useful containers (collections). Two notable ones:

- `std::vector<T>`: A dynamic array (à la `ArrayList<T>` in Java).
- `std::list<T>`: A doubly linked list, which can be used as a queue or deque.

Both can be initialized as so:

```
#include <list>;

std::list<int> a; // create an empty list
std::list<int> b { 1, 2, 3 }; // create a list [1,2,3]
```

For this project, if you need a double-ended queue, you should generally use `std::list` and avoid using `std::queue` or `std::deque`. It is almost always simpler and less headache-inducing to do so.

The performance difference between `std::list` and `std::deque` is negligible for this project, and you are likely to run into memory issues with `std::deque` (unless you are fully aware of iterator invalidation rules). You have been warned.

Iterators

Note: If you're coming from a language like Python or Rust, know that C++ iterators do not have many of the functional programming operations that you would expect with iterators from those languages.

Iterators are pointers to container elements with special powers that allow you to effectively iterate through the elements of a container (woah!). Like pointers, you can use the `*` operator to read and write to the data pointed to by an iterator.

STL containers provide iterators through the `.begin()` and `.end()` methods, where `.begin()` points to the first element of the container, and `.end()` points to one past the last element of the container.

To use an iterator in a for-loop, you can use any of the following patterns:

```
std::list<int> data { 1, 2, 3, 4 };
for (auto it = data.begin(); it != data.end(); ++it) {
    printf("%d\n", *it); // do something with the data
}

// Syntax sugar for the above. This works on any type which defines an iterator.
for (auto v : data) {
    printf("%d\n", v);
}

std::vector<BigUncopiableType> big_data { ... };
// For structs that aren't trivial to copy, be sure to take each element by reference:
for (auto& v : big_data) {
    printf("%d\n", v.x);
}
```

Note: `auto` means to infer the type of this variable.

If you wanted to create a "find element" function, you could do the following:

```
std::list<int>::iterator find_element(std::list<int> &data) {
    for (auto it = data.begin(); it != data.end(); i++) {
        if (predicate(*it)) return it;
    }

    // If we don't find the element, use the end pointer
    // (which doesn't point to an element) as an error value.
    return data.end();
}
```

The usefulness of returning an iterator here (instead of a reference to the element) is that we can use the `data.end()` iterator to indicate the element doesn't exist. So, for instance, we can use this function as follows:

```
auto item = find_element(data);
if (item != data.end()) {
    // now we know "item" exists and we can dereference it!
}
```