CS {4/6}290 & ECE {4/6}100 - Spring 2024
Project 1: Cache Simulator
Version 1.1

Dr. Thomas Conte and Pulkit Gupta
Due: February 16 2024 @ 11:55 PM

1 Changelog

- **Version 1.1**: 2024-02-01: Section 4.1 change L1 associativity restriction from $S_{L1} \geq 1$ to $S_{L1} \geq 0$. All files from the new tarball except for `cachesim.cpp` have changed. Clarified grad section requirements.
- **Version 1.0**, 2024-01-30: 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 Head TA/Instructors.
- 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 **C11 or C++20** standards, using only the standard libraries.

3 Background

3.1 Prefetcher (+1 and Strided)

In this project, you will build a simulator with a write-back, write-allocate L1 cache, and a write through, write-no-allocate L2 cache. You will only implement a prefetcher in the L2 cache. Remember that L2 cannot miss on write due to being write non-allocate. You only

prefetch a block when an L2 cache **read miss** happens and the block you want to bring in is not yet in the cache.

In this project we will use two prefetching schemes. The first one being a simple +1 Prefetcher and the second one being a Strided Prefetcher. Since the minimum unit the Prefetcher can fetch is a block, we only use the block address and ignore the block offset. Where "block address" is the concatenation of the tag and the index.

> **+1 Prefetcher**: On a miss on block at block address X, if the next block (block at block address $(X + 1)$) is not in the cache, prefetch that block and insert it into the L2 cache. Prefetch $(X + 1)$ only after inserting block X into the cache.
>
> **Strided Prefetcher**: On a miss on a block at block address X, then on a block at block address Y, prefetch the block at block address $Y + k$ where $k = Y - X$, if block at address $Y + k$ is not in the cache. k is the difference between the current miss and the previous. Prefetch $(Y + k)$ only after the block at block Y is inserted into the cache.

3.1.1 Insertion Policies (MIP and LIP)

In this project, we will consider two cache insertion policies when dealing with prefetching. The first is the standard insertion policy used in LRU replacement, which we call "MIP" for clarity:

> **MRU Insertion Policy (MIP)**: The MRU insertion policy means that new blocks are inserted in the MRU position. MRU insertion policy is originally defined only under the LRU replacement policy. For this project, we extend the definition of MIP under the LFU replacement policy. In this project, we define the MIP under LFU as setting the MRU bit of a block and clear other the MRU bit of every other block in the same set.
>
> **LRU Insertion Policy (LIP)**:
>
> Designed for L2 caches to avoid thrashing on workloads with low temporal locality, LIP inserts all new blocks at the LRU position. Similarly, for this project, we define of LIP under the LFU replacement policy. We define LIP under LFU as clearing the MRU bit of the inserted block and leaving the MRU bit of other blocks in the set unmodified.

4 Simulator Specifications

In this project, you will build a simulator with a write-back, write-allocate L1 cache, and a write through, write-no-allocate L2 cache. Your simulator will receive a series of memory accesses from the CPU in a provided trace and must print some final statistics before exiting.

4.1 Simulator Configuration

Both the L1 and L2 caches should implement LRU replacement (**for everyone**) and LFU replacement (**for students in 6100 & 6290**). The L1 cache follows the write-back, write-allocate write strategy. The L2 cache shares the L1 block size (B parameter) and follows the write-through, write-no-allocate write strategy. The system uses 64-bit addresses.

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

- -c $< C_{L1} >$: A total size of $2^{C_{L1}}$ bytes
- -b $< B >$: Block size of 2^B bytes. This will also be the block size used for the L2 cache

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

- -s < S_{L1} >: $2^{S_{L1}}$-way associativity

  Restriction: $S_{L1} \geq$ ~~1~~ 0

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

- -D: if this flag is supplied, the L2 cache is Disabled, it is enabled otherwise
- -C < C_{L2} >: A total size of $2^{C_{L2}}$ bytes

  Restriction: The size of the L2 cache must be strictly greater than the size of the L1 cache ($C_{L2} > C_{L1}$)

- -S < S_{L2} >: $2^{S_{L2}}$-way associativity

  Restriction: The associativity of the L2 cache must be greater than the associativity of the L1 cache ($S_{L2} > S_{L1}$)

  Restriction: The L2 $C_{L2} - S_{L2}$ must be greater than the L1 $C_{L1} - S_{L1}$

- -P < $0, 1, 2$ >: Sets the type of prefetcher to use
  - 0: Disable prefetcher
  - 1: Use +1 prefetcher **Everyone**
  - 2: Use Strided Prefetcher **students in 6100 & 6290**
- -I <MIP, LIP>: Prefetcher Insertion Policy

The following parameters apply to the entire simulation:

- -r <LRU, LFU>: Replacement policy for both caches: LRU **(Everyone)** or LFU **(LFU is for students in 6100 & 6290)**
- -f <Filename>: The filename/path for the trace to execute

4.2 Simulator Semantics

4.2.1 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. Also, 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. Note that a write miss in L1 will first read the block from L2 and then update the L1 copy with the write.

4.2.2 Prefetching

Prefetching in the cache system is a trick that uses spatial/sequential locality to try to predict what blocks might be needed in the future. The main two types of prefetching schemes that we are going to implement in this cache system are the +1 prefetch and the strided prefetch.

+1 Prefetching
On an L2 cache read miss, we fetch the missing block from memory, but also fetch the next block in memory and bring it into the cache. If the next block is already in cache, we skip the prefetch.

Strided Prefetching (for students in 6100 & 6290)
On an L2 cache read miss, we not only fetch the missing block from memory, but also fetch the block that is k block addresses away where k equals the difference between the current block address and the previous miss's block address block address. For every subsequent L2 read miss, make sure to use the most recent previous L2 read miss to calculate your stride. To implement the strided prefetcher, you keep track of "previous miss". You initialize "previous miss" to **0x0**. For the first L2 read miss, the prefetch address you calculated from that "previous miss" may be meaningless. However, if that block at the calculated address is not in the cache, you still prefetch that block and increase `prefetches_l2` by 1.

4.2.3 LRU Replacement and Tracking

One possible way to implement the LRU Replacement Policy is to use timestamp counters. **We strongly recommend using timestamps to implement LRU tracking for this project as it follows the same logic of our implementation.** To use this implementation, you need to keep track of the counter value where a certain block has been used and update it every time that block is used. This needs to be done **per cache** as there is possible cases where the LRU is different in the L1 cache and in the L2 cache. When it comes to evict a block, simply look through the timestamps associated with each block and evict the one with the farthest back (smallest) timestamp.

Here is how you will use and manage the timestamp counter.

- First, you initialize the cache's timestamp counter to the value $2^{(C-B+1)}$.
- In cases where the block of current access or the block you want to prefetch is already in the cache:
  - If the block of current access is already in cache, you set the timestamp of the block to the current value of the timestamp counter. This acts like setting the block to most recently used. After access, you increment the timestamp counter by 1.
  - **Case where you do not change the timestamp:** If the block you want to prefetch is already in the cache, prefetching does not happen, and you do nothing. You do not increment the timestamp counter.
- In cases where you need to install a new block, there are two possible insertion policies: MIP or LIP. The block of current access can only use MIP, and the block you want to prefetch may use MIP or LIP based on configuration.
  - For MIP, you set the timestamp of the newly installed block to the current value of the timestamp counter. After insertion, you increment the timestamp counter by 1.
  - For LIP, you set the timestamp of the newly installed block to $lowest - 1$. After insertion, you increment the timestamp counter by 1. We define *lowest* as the smallest timestamp of all valid blocks in the set you want to install a new block into. **If**

there is no valid block in the set before insertion, *lowest* is undefined. In this case where *lowest* is undefined, you set the timestamp of the newly installed block to current value of the timestamp counter.

4.2.4 LFU Replacement and Tracking

Note: Implementation of LFU Replacement Policy is for students in 6100 & 6290. To implement the LFU Replacement Policy, you must keep a counter of how many times a certain block is used and refer to the counter when deciding which block to evict. However, the naive LFU Replacement Policy has one huge fatal flaw. Imagine missing on a block with a full cache. When you bring in that new block, you replace the LFU, but now that new block is most likely the LFU. If the next access is also a miss, then that next block will replace the block that you just brought in. This is very inefficient and goes against temporal locality, therefore we will use the MRU bit to prevent this case. The MRU bit serves as a way to prevent a block from being evicted. When it comes to evict a block, simply check each counter for each block and evict the one with the lowest counter with the restriction that you do not evict the block that has the MRU bit set. In the cases where multiple blocks are equal in terms of frequency of use, break the tie by choosing to evict the block with the **smallest** tag.

We track LFU status by having a "use" counter for each block that counts how many times it has been used and a MRU bit for each block that tells us if it is the MRU block in the set.

- In cases where the block of current access or the block you want to prefetch is already in the cache:
  - If the block of current access is already in cache, you set the MRU bit of the block **and clear the MRU bit of every other block in the same set**. After access, you increment the block's "use" counter by 1.
  - **Case where you leave the MRU bit and the "use" counter alone:** If the block you want to prefetch is already in cache, prefetching does not happen, and you do nothing. You do not increment the "use" counter nor touch the MRU bit.
- In cases where you need to install a new block, there are two possible insertion policies: MIP or LIP. The block of current access can only use MIP, and the block you want to prefetch may use MIP or LIP based on configuration:
  - For MIP, you set the MRU bit of this new block **and clear the MRU bit of every other block in the same set**.
  - For LIP, you clear the MRU bit of this new block. You leave the MRU bit of the other blocks in the set unmodified.
- If the newly installed block is the block of current access, set the "use" counter to 1. If the newly installed block is the block you prefetched, set the "use" counter to 0.

4.2.5 Serial Checks

Although there may be plenty of parallelism in a real-life cache hierarchy, you will implement memory accesses in the simulator serially because it helps calculate the statistics we care about in this project. The sequence of checks for a given memory access should be:

1. Check L1 cache

2. Check L2 cache

However, your simulator should increment hit/miss counters for the L2 cache only when L1 has missed. Similarly, blocks in L2 should not move into the MRU position for an access unless the L1 cache missed.

4.2.6 Do Not Writeback Before L2 Read

When the L1 cache misses, please have the L1 cache perform a read from L2 before you write back a dirty victim block to L2. Otherwise, if the block 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.

4.2.7 Disabling Caches

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.

4.3 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 4.3.1
- 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 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 cache
  - L2 read hits (`read_hits_l2`): Total number of L2 read hits. This stat should not be touched unless there was an L1 cache miss
  - L2 read misses (`read_misses_l2`): Total number of L2 read misses. This stat should not be touched unless there was an L1 cache miss

- L2 prefetches (`prefetches_l2`): Total number of L2 prefetches. This stat should not be touched unless the block you want to prefetch is not in the cache at the time you want to bring it in.
- 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 4.3.1

4.3.1 Average Access Time

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

- For L1, hit time is $k_0 + (k_1 \times (C_{L1} - B_{L1} - S_{L1})) + k_2 \times (max(3, S_{L1}) - 3)$
  - $k_0 = 1\,\text{ns}$
  - $k_1 = 0.15\,\text{ns}$
  - $k_2 = 0.15\,\text{ns}$
- For L2, hit time is $k_3 + (k_4 \times (C_{L2} - B_{L2} - S_{L2})) + k_5 \times (max(3, S_{L2}) - 3)$
  - $k_3 = 4\,\text{ns}$
  - $k_4 = 0.3\,\text{ns}$
  - $k_5 = 0.3\,\text{ns}$
- The time to access DRAM is 100 ns

The provided header file, `cachesim.hpp`, includes constants for these values. These values are in nanoseconds. Please also provide your answer in nanoseconds.

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

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

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

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.

5 Implementation Details

We included 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 4.1; 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.

5.1 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 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.

6 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`), L1 and L2 with prefetcher disabled (`-P 0`). 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` **(for students in 4100 & 4290)** and `make validate_grad` **(for students in 6100 & 6290)**.

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.

6.1 Debug Traces

Debugging this project can be difficult, particularly when looking only at final statistics. We have provided some verbose debug traces for you that correspond to the reference traces. To motivate you to use them, **part of your grade will depend on you implementing matching debug traces in your own code** (see Section 9). Please see the `debug_outs/README.txt` for more information.

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.

7 Experiments

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

1. Configuration should respect all the restrictions mentioned in Section 4.1
2. The L2 data store is 128 KiB (C = 17)
3. The L1 data store is 32 KiB (C = 15)

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

In your report you must provide, in addition to the cache configuration, the total size of any associated metadata required to build the cache for your recommended configuration(s), except for the LRU/LFU tracking metadata.

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/LFU information has no implementation cost for the simplicity of the project.

You should submit a report in PDF format (inside your submission tarball) in which you describe your methodology, rationale, and results.

8 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 and test your implementation.

9 Grading

You will be evaluated on the following criterion:

+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

+35: Your simulator **completely matches** the validation output

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

+5: You implement debug outputs as explained in Section 6.1. Your code is well formatted, commented and does not have any memory leaks! Check out the section on helpful tools

Appendix A - Plagiarism

We take academic plagiarism very seriously in this course. Any and all cases of plagiarism are reported to the Dean of Students. We use accepted forensic techniques to determine whether

there is copying of a coding assignment. You may not do the following in addition to the Georgia Tech Honor Code:

- Copy/share code from/with your fellow classmates or from people who might have taken this course in prior semesters.
- Publish your assignments on public repositories, github, etc, that are accessible to other students.
- Submit an assignment with code or text from an AI assistant (e.g., ChatGPT).
- Look up solutions online. Trust us, we will know if you copy from online sources.
- Debug other people's code. You can ask for help with using debugging tools (Example: Hey XYZ, could you please show me how GDB works), but you may not ask or give help for debugging the cache simulator.
- You may not reuse any code from earlier courses even if you wrote it yourself. This means that you cannot reuse code that you might have written for this class if you had taken it in a prior semester. You must write all the code yourself and during this semester.

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 your own conversation, but do not copy anything from the conversation back into your assignment. Instead, use your interaction with the AI assistant as a learning experience, 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.

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 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>` 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>
```