CS {4/6}290 & ECE {4/6}100 - Spring 2022
Project 1: Cache Simulator
Version 1.2

Professor Tom Conte

Due: February 14th 2022 @ 11:55 PM

1 Changelog

- **Version 1.2**, 2022-01-30: The TA solution had a bug: on a victim cache hit, when inserting the victim cache block into L1 cache, our code set the dirty bit in L1 using the R/W bit of the L1 request instead of ORing that with the dirty bit on the victim cache entry. The reference and debug outputs have been updated
- **Version 1.1**, 2022-01-29: For clarity, introduce the term MIP in Section 3.1 to better contrast with LIP. The term MIP is now used in the provided base code rather than saying "LRU," which could be confused with LIP
- **Version 1.0**, 2022-01-28: 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[1].

[1] If you choose to write your project from scratch in Java (highly discouraged), please use Java 11 and only the standard library for Java 11.

3 Background

3.1 Insertion Policies (MIP and LIP)

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

> **MRU Insertion Policy (MIP)**: This is the standard insertion policy used in LRU replacement. New blocks are inserted in the MRU position.

Contrast this with the following insertion policy proposed by Qureshi et al. [2]:

> **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.

3.2 Victim Cache

First proposed by Jouppi [1], a victim cache offers some on-demand extra associativity to the L1 cache. When evicting a block from L1, that victim block is inserted into the victim cache (hence the name). A victim cache is always fully-associative and always uses LRU replacement (with MIP), so when the victim cache is already full when trying to insert an L1 victim into the victim cache, the LRU block in the victim cache should be evicted. If the block evicted from the victim cache is dirty, it should be written back to L2.

When the L1 cache misses, the victim cache is searched for requested block[2]; if found, the block in the victim cache is swapped with a victim block in the L1 cache chosen according to the L1 replacement policy. Both swapped blocks are set to the MRU position in their respective caches.

4 Simulator Specifications

In this project, you will build a simulator for the cache hierarchy shown in Figure 1, specifically the parts shown within the dashed lines: a write-back, write-allocate L1 cache with a victim 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.

[2]In reality, both the victim cache and L1 cache would be searched in parallel. But for the purpose of this project, assume the victim cache is searched only after an L1 miss.

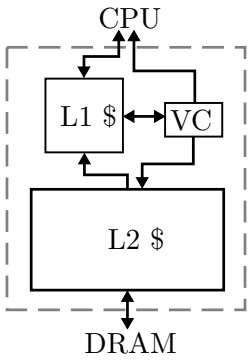

Figure 1: Cache hierarchy simulated in this project. "VC" stands for victim cache.

4.1 Simulator Configuration

Both the L1 and L2 caches should implement LRU replacement. The L1 cache follows the write-back, write-allocate write strategy, and L1 always uses MIP. The victim cache is always fully associative, always uses LRU replacement policies and MIP, and always has the same block size (B parameter) as the L1 cache. 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: A total size of 2^C bytes
- B: Block size of 2^B bytes. This will also be the block size used for the victim cache and L2

  Restriction 2: The block size must be reasonable: $4 \leq B \leq 7$
- S: 2^S-way associativity

The L1 cache has an associated victim cache (see Section 3.2) with only one knob:

- Number of victim cache entries, that is, the number of blocks it can hold. Passing zero disables the victim cache

  Restriction: The number of victim cache entries must be 0, 1, or 2

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

- Enabled/Disabled
- C: A total size of 2^C bytes

  Restriction: The size of the L2 cache must be strictly greater than the size of the L1 cache
- S: 2^S-way associativity

  Restriction: The associativity of the L2 cache must be greater than or equal to the associativity of the L1 cache
- Insertion policy: MIP or LIP

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 (plus its associated victim 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.

4.2.2 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 victim cache
3. Check L2 cache

For example, in real hardware, one would send an access request to both the L1 cache and victim cache in parallel. However, your simulator should increment hit/miss counters for the victim cache only when L1 has missed, and only increment hit/miss counters for L2 when the victim cache misses, and so on. Similarly, blocks in L2 should not move into the MRU position for an access unless both L1 and the victim cache missed.

4.2.3 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.

Similarly, when the victim cache is disabled (by setting its number of entries to zero), your simulator should update statistics as if it was empty as well: that is, always increment the number of victim cache misses for every L1 miss.

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. Victim cache hits should not be included in this statistic
  - L1 misses (`misses_l1`): Total number of L1 misses. This must be incremented even if the block is in the victim cache
  - L1 hit ratio (`hit_ratio_l1`): Hit ratio for L1, calculated using the first three L1 stats above. (Consequently, victim cache hits do not count as L1 hits here)
  - L1 miss ratio (`miss_ratio_l1`): Miss ratio for L1, calculated using the first three L1 stats above. (Consequently, L1 misses count even if the block is in the victim cache)
  - L1 AAT (`avg_access_time_l1`): See Section 4.3.1
- Victim cache statistics:

  - Victim cache hits (`hits_victim_cache`): Number of hits in the victim cache after an L1 miss (see Section 4.2.2)
  - Victim cache misses (`misses_victim_cache`): Number of hits in the victim cache after an L1 miss (see Section 4.2.2)
  - Victim cache hit ratio (`hit_ratio_victim_cache`): Hit ratio for the victim cache, calculated using the first two victim cache stats and `misses_l1`
  - Victim cache miss ratio (`miss_ratio_victim_cache`): Miss ratio for the victim cache, calculated using the first two victim cache stats and `misses_l1`
  - Write-backs from L1 or victim cache (`write_backs_l1_or_victim_cache`): Write-backs performed by the victim cache, or by the L1 cache when the victim cache is disabled; that is, the number of dirty blocks evicted from the victim cache, or from L1 when the victim cache is disabled

- 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 both an L1 and victim cache 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 or the victim cache
  - L2 read hits (`read_hits_l2`): Total number of L2 read hits. This stat should not be touched unless there was both an L1 and victim cache miss
  - L2 read misses (`read_misses_l2`): Total number of L2 read misses. This stat should not be touched unless there was both an L1 and victim cache 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 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 $2\,\text{ns} + (S_{\text{L1}} \times 0.2\,\text{ns})$
- For L2, hit time is $8\,\text{ns} + (S_{\text{L2}} \times 0.8\,\text{ns})$
- The time to access DRAM is $80\,\text{ns}$

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 slides:

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

where the hit time HT_{L1} is as calculated above[3], as it could be accessed in parallel), MR_{VC} is the miss ratio of the victim cache after an L1 miss (`miss_ratio_victim_cache`), and MR_1 is the traditional L1 miss ratio (`miss_ratio_l1`). Think: in this project, what is the miss penalty of the L1+victim cache combination ($\text{MP}_{\text{L1+VC}}$ in the equation above)?

[3]We assume no additional time is needed to check the victim cache, as it would be accessed in parallel with the L1 cache in hardware.

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. However, it is allowed as long as `make && ./run.sh <cachesim args> < traces/X.trace` (equivalent to `make && ./cachesim <cachesim args> < traces/X.trace` if you have not changed `./run.sh`) produces byte-for-byte matching outputs as the solution outputs.

5.1 Implementation in Java (Strongly Discouraged)

If you make the highly discouraged choice to write your project from scratch in Java 11, you need to set up the `Makefile` with a default target that will compile your code with `javac`. Please also include a `clean` target that will remove `.class` files. You will also need to set up `./run.sh` to run your Java simulator, which must accept all the same configuration flags as `cachesim_driver.cpp` (`-c`, `-b`, `-s`, etc.). Your code must not depend on any libraries or functions not included in the standard library for Java 11.

5.2 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.

Instead, we suggest using either a C++ `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.

5.3 Docker Image

We have provided an Ubuntu 18.04 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 18.04 `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 18.04. 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 (`-v 0 -D`), only L1 and victim cache (`-D`), and only L1 and L2 (`-v 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`.

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, so we have also provided some verbose debug traces corresponding to the reference traces in a separate tarball on Canvas:
`project1_vX_Y_debug_outs.tar.xz`. 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**.

7 Experiments

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 4.1
2. The budget for the L2 data store is 128 KiB
3. The budget for the L1 data store is 32 KiB
4. The budget for the Victim Cache data store is 128 bytes

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

You should submit a report in PDF format (inside your submission tarball) in which you describe your methodology 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 18.04 at submission time. The Gradescope autograder result is NOT indicative of your grade on the project and will only check that the assignment compiles.

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

+25: Your simulator **completely matches** the validation output

+20: You ran experiments and found the optimum configuration for the 'experiments' workload and presented sufficient evidence and reasoning

+5: 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. 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.
- 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.

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> < 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
```

References

[1] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[2] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381–391, June 2007.