CS 211: Computer Architecture, Spring 2024
Programming Assignment 5: Simulating Caches (100 points)

Instructor: Prof. Santosh Nagarakatte

Due: April 25, 2024 at 5pm Eastern Time.

Introduction

The goal of this assignment is to help you understand caches better. You are required to write a cache simulator using the C programming language. The programs have to run on iLab machines. We are providing real program memory traces as input to your cache simulator.

No cheating or copying will be tolerated in this class. Your assignments will be automatically checked with plagiarism detection tools that are powerful. Hence, you should not look at your friend's code or use any code from the Internet or other sources such as Chegg/Freelancer. We strongly recommend not to use any large language model and use the code sample provided by it as it will likely trigger plagiarism violations. All violations will be reported to office of student conduct. See Rutgers academic integrity policy at:
`http://academicintegrity.rutgers.edu/`

Memory Access Traces

The input to the cache simulator is a memory access trace, which we have generated by executing real programs. The trace contains memory addresses accessed during program execution. Your cache simulator will have to use these addresses to determine if the access is a hit or a miss, and the actions to perform in each case. The memory trace file consists of multiple lines. Each line of the trace file corresponds to a memory access performed by the program. Each line consists of two columns, which are space separated. First column lists whether the memory access is a read (R) or a write (W) operation. The second column reports the actual 48-bit memory address that has been accessed by the program. Here is a sample trace file.

```
R 0x9cb3d40
W 0x9cb3d40
R 0x9cb3d44
W 0x9cb3d44
R 0xbf8ef498
```

Part One - One Level Cache - 50 points

You will implement a cache simulator to evaluate different configurations of caches. The followings are the requirements for the first part of the cache simulator.

- Simulate only one level cache; *i.e.*, an L1 cache.
- The cache size, associativity, the replacement policy, and the block size are input parameters. Cache size and block size are specified in bytes.
- You have to simulate a write through cache.
- Replacement Algorithm: You have to support two replacement policies. The two replacement policies are: First In First Out (FIFO) and Least Recently Used (LRU).

Next, you will learn more about cache replacement policies.

Cache Replacement Policies

The goal of the cache replacement policy is to decide which block has to be evicted in case there is no space in the set for an incoming cache block. It is always preferable – to achieve the best performance – to replace the block that will be re-referenced furthest in the future. In this assignment, you will use two different ways to implement the cache replacement policy: FIFO and LRU.

FIFO

When the cache uses the FIFO replacement policy, it always evicts the block accessed first in the set without considering how often or how many times the block was accessed before. So let us say that your cache is empty initially and that each set has two ways. Now suppose that you access blocks A, B, A, C. To make room for C, you would evict A since it was the first block to be brought into the set.

LRU

When the cache used the LRU replacement policy, it discards the least recently used items first. The cache with an LRU policy has to keep track of all accesses to a block and always evict the block that been used (or accessed) least recently as the name suggests.

Cache Simulator Interface

You have to name your cache simulator first. Your program should support the following usage interface:

```
./first <cachesize> <assoc:n> <cache policy> <block size> <trace file>
```

where:

- The parameter **cache size** is the total size of the cache in bytes. This number should be a power of 2.

- The parameter **assoc:n** specifies the associativity. Here, n is a number of cache lines in a set.
- The parameter **cache policy** specifies the cache replacement policy, which is either fifo or lru.
- The parameter **block size** is a power of 2 that specifies the size of the cache block in bytes.
- The parameter **trace file** is the name of the trace file.

Simulation Details

- When your program starts, there is nothing in the cache. So, all cache lines are empty.
- You can assume that the memory size is 2^{48} . Therefore, memory addresses are at most 48 bit (zero extend the addresses in the trace file if they are less than 48-bit in length).
- The number of bits in the tag, cache address, and byte address are determined by the cache size and the block size.
- For a write-through cache, there is the question of what should happen in case of a write miss. In this assignment, the assumption is that the block is first read from memory (*i.e.*, one memory read), and then followed by a memory write.
- You do not need to simulate data in the cache and memory in this assignment. Because, the trace does not contain any information on data values transferred between memory and caches.

Sample Run

Your program should print out the number of memory reads (per cache block), memory writes (per cache block), cache hits, and cache misses. You should follow the exact same format shown below (no space between letters), otherwise, the autograder can not grade your program properly.

```
./first 32 assoc:2 fifo 4 trace1.txt
memread:336
memwrite:334
cachehit:664
cachemiss:336
```

The above example, simulates a 2-way set associate cache of size 32 bytes. Each cache block is 4 bytes. The trace file name is trace1.txt.

Note: Some of the trace files are quite large. So it might take a few minutes for the autograder to grade all testcases.

Part II - Two Level Cache - 50 points

Most modern CPUs have multiple level of caches. In the second part of the assignment, you have to simulate a system with a two-level of cache (*i.e.*, L1 and L2). Multi-level caches can be designed in various ways depending on whether the content of one cache is present in other levels or not. In this assignment you implement an exclusive cache: the lower level cache (*i.e.*, L2) contains only blocks that are not present in the upper level cache (*i.e.*, L1).

Exclusive Cache

Consider the case when L2 is exclusive of L1. Suppose there is a read request for block X. If the block is found in L1 cache, then the data is read from L1 cache. If the block is not found in the L1 cache, but present in the L2 cache, then the cache block is moved from the L2 cache to the L1 cache. If this causes a block to be evicted from L1, the evicted block is then placed into L2. If the block is not found in either L1 or L2, then it is read from main memory and placed just in L1 and not in L2. In the exclusive cache configuration, the only way L2 gets populated is when a block is evicted from L1. Hence, the L2 cache in this configuration is also called a victim cache for L1.

Sample Run

The details from Part 1 apply here to the second level L2 cache. Your program gets two separate configurations (one for level 1 and one for level 2 cache). Both L1 and L2 have the same block size. Your program should report the total number of memory reads and writes, followed by cache miss and hit for L1 and L2 cache. Here is the format for part 2.

```
./second <L1 cache size> <L1 associativity> <L1 cache policy> <L1 block size>
 <L2 cache size> <L2 associativity> <L2 cache policy> <trace file>
```

This is an example testcase for part 2.

```
./second 32 assoc:2 fifo 4 64 assoc:16 lru trace2.txt
memread:3277
memwrite:2861
l1cachehit:6501
l1cachemiss:3499
l2cachehit:222
l2cachemiss:3277
```

The above example, simulates a 2-way set associate cache of size 32 bytes. bytes with block size of 4 for L1 cache. Similarly, L2 cache is a fully associate cache of size 64 bytes. Further, the trace file used for this run is trace2.txt. As you can see, the program outputs the memory read and memory writes followed by the L1 and L2 cache hits and misses in the order shown above.

Structure of your submission

All files must be included in the pa5 folder. The pa5 directory in your tar file must contain 2 subdirectories, one each for each of the parts. The name of the directories should be named first and second. Each directory should contain a c source file, a header file (optional) and a Makefile. To create this file, put everything that you are submitting into a directory named pa5. Then, cd into the directory containing pa5 (*i.e.*, pa5's parent directory) and run the following command:

```
tar cvf pa5.tar pa5
```

To check that you have correctly created the tar file, you should copy it (pa5.tar) into an empty directory and run the following command:

```
tar xvf pa5.tar
```

This is how the folder structure should be.

```
* pa5
 - first
    * first.c
    * first.h
    * Makefile
 - second
    * second.c
    * second.h
    * Makefile
```

Autograder

First Mode

Testing when you are writing code with a pa5 folder.

- Let us say you have a pa5 folder with the directory substructure as described in the assignment description.
- copy the pa5 folder to the directory of the autograder.
- Run the autograder with the following command.

  ```
  python3 pa5_autograder.py
  ```

It will run the test cases and print your scores.

Second Mode

This mode is to test your final submission (*i.e.*, pa5.tar)

- Copy pa5.tar to the autograder directory.
- Run the autograder with pa5.tar as the argument as shown below

```
python3 pa5_autograder.py pa5.tar
```

Grading Guidelines

This is a large class so that necessarily the most significant part of your grade will be based on programmatic checking of your program. That is, we will build the binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- You should not see or use your friend's code either partially or fully. We will run state of the art plagiarism detectors. We will report everything caught by the tool to Office of Student Conduct.
- You should make sure that we can build your program by just running make.
- Your compilation command with gcc should include the following flags: -Wall -Werror -fsanitize=address
- You should test your code as thoroughly as you can. For example, programs should not crash with memory errors.
- Your program should produce the output following the example format shown in previous sections. Any variation in the output format can result in up to 100% penalty. Be especially careful to not add extra whitespace or newlines. That means you will probably not get any credit if you forgot to comment out some debugging message.
- Your folder names in the path should have not have any spaces. Autograder will not work if any of the folder names have spaces.

Be careful to follow all instructions. If something doesn't seem right, ask on Canvas discussion forums or contact the TAs during office hours.