CS 211 Fall 2019
Assignment 4 - Cache Simulator
Instructor : Prof. Santosh Nagarakatte

Due : Nov 27, 5:00 PM

1 Overview

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. The format and structure of the memory traces are described below.

2 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 accesses performed by the program. Each line consists of two columns, which are space separated. The second column reports 48-bit memory address that has been accessed by the program while the first column indicates whether the memory access is a read (R) or a write (W). The trace file always ends with a #eof string. We have provided you three input trace files (some of them are larger in size). You can safely assume the trace files always have proper format.
Here is a sample trace file.

```
R 0x9cb3d40
W 0x9cb3d40
R 0x9cb3d44
W 0x9cb3d44
R 0xbf8ef498
#eof
```

3 Cache Simulator

You will implement a cache simulator to evaluate different configurations of caches. Your program should be able to support traces with any number of lines. The followings are the requirements for your cache simulator:

- You simulate one level cache.
- The cache size, associativity, the replacement policy, and the block size are the input parameters. Cache size and block size are specified in bytes.
- Replacement algorithm: First In First Out (FIFO).
- You have to simulate a write through cache.

4 Cache Simulator Interface

You have to name your cache simulator C code **first**. Your program should support the following usage interface: ./first <cache size><block size><cache policy><associativity><prefetch size><trace file>

where:
A) <cache size>is the total size of the cache in bytes. This number should be a power of 2.
B) <block size>is a power of 2 integer that specifies the size of the cache block in bytes.
C) <cache policy>Here is valid cache policy is *fifo* and *lru* if you do the extra credit.
D) <associativity>is one of:
direct - simulate a direct mapped cache.
assoc - simulate a fully associative cache.
assoc:n - simulate an n way associative cache. n will be a power of 2.
E) <prefetch size>is the number of ajacent blocks that should be prefetched by the prefetcher in case of a miss
F) <trace file>is the name of the input trace file.

NOTE: Your program should check if all the inputs are in valid format, and the trace file exist. if not print **error** and then silently terminate the program.

5 Cache Prefetcher

Prefetching is a common technique to increase the spatial locality of the caches beyond the cache line. The idea of prefetching is to bring the data into the cache before it is needed (accessed). In a normal cache, you bring a block of data into the cache whenever you

experience a cache-miss. Now, we want you to explore a different type of cache that prefetches not only brings the block corresponding to the access but also prefetches some adjacent blocks. The number of adjacent blocks that has to be prefetched every time prefetcher enabled, is provided by the user as an input.
Here is some important assumptions about the prefetcher. First, the prefetcher is activated **only** on cache misses. Second, none of the counters (including the memory read) are updated if the prefetched block already exist in the cache. Third, with respect to cache replacement policies, if the prefetched block hits in the cache, the block replacement policy status should **not** be updated. Otherwise, it is treated similar to a block that missed the cache.

6 Sample Output

In your program, you should keep track of four counters. These four counters are memory reads (per cache block), memory writes (per cache block), cache hits, and cache misses. As the output, your program should print out these four counter in the format shown below. You should follow the exact same format (pay attention to case sensitivity of the letters), otherwise, the autograder can not grade your program properly and you do not get any points.

```
$./first 32 4 fifo assoc:2 1 trace2.txt
no-prefetch
Memory reads: 3499
Memory writes: 2861
Cache hits: 6501
Cache misses: 3499
with-prefetch
Memory reads: 3521
Memory writes: 2861
Cache hits: 8124
Cache misses: 1876
```

In this example, we are simulating a 2-way set associate cache of size 32 bytes. Each cache block is 4 bytes. The prefetch size is 1, meaning that we prefetch only one adjacent block. The trace file name is trace2.txt.

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

7 Cache Replacement Policy

he 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.

There are different ways one can implement cache replacement policy. Here we use **FIFO** replacement policy and **LRU** policy is extra credit.

7.1 FIFO

Using this algorithm, you can always evict the block accessed first in the set without any regard to how often or how many times it 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.

7.2 LRU

Least Recently Used (LRU) replacement policy is used replace the cache line that is least recently used. Let us use the same example as the one for FIFO. If we access A,B,A,C. To make room for C, we should evict B since it is the least recently used block (in contrast to FIFO which we evicted A)

8 Other Details

1. (a) When your program starts, there is nothing in the cache. So, all cache lines are empty (invalid).
(b) you can assume that the memory size is 2pow48 . Therefore, memory addresses are 48 bit (zero extend the addresses in the trace file if theyre less than 48-bit in length).
(c) the number of bits in the tag, cache address, and byte address are determined by the cache size and the block size;

2. 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 (one read memory), and then followed by a memory write.
3- You do not need to simulate the memory in this assignment. Because, the traces doesnt contain any information on data values transferred between the memory and the caches.
4. You have to compile your program with the following flags:
-Wall -Werror -fsanitize=address
5. You should include a makefile in you submission.

9 Extra credit (25 points)

As an extra credit, you should implement LRU (Least Recently Used) cache policy. Your program should output exactly the same format output as it shown before. Here is an example of running your program with LRU policy.

```
$./first 32 4 lru assoc:2 1 trace2.txt
```

10 Submission

You have to e-submit the assignment using Sakai . Put all files (source code + Makefile) into a directory named **first**, which itself is a sub-directory under pa4 . Then, create a tar file (follow the instructions in the previous assignments to create the tar file). Your submission should be only a tar file named pa4.tar. You have to e-submit the assignment using Sakai. Your submission should be a tar file named pa4.tar. To create this file, put everything that you are submitting into a directory named pa4. Then, cd into the directory containing pa4 (that is, pa4s parent directory) and run the following command:

```
$tar cvf pa4.tar pa4
```

To check that you have correctly created the tar file, you should copy it (pa4.tar) into an empty directory and run the following command:

```
$tar xvf pa4.tar
```

This is how the folder structure should be.

- pa4
  - first
    - first.c
    - first.h
    - Makefile

Source code: all source code files necessary for building your programs. e.g. first.c and first.h.
Makefile: There should be at least three rules in your Makefile:
all: make a complete build of your program (first).
first: build the executables (first).
clean: prepare for rebuilding from scratch.

Do **NOT** INCLUDE ANY OF THE TRACE FILES IN THE FOLDER

11 Autograder

First mode
Testing when you are writing code with a pa4 folder.

1. Lets say you have a pa4 folder with the directory structure as described in the assignment.
2. Copy the folder to the directory of the autograder
3. Run the autograder with the following command
$python auto_grader.py
It will run the test cases and print your scores.
Second mode
This mode is to test your final submission (i.e, pa4.tar) 1. Copy pa4.tar to the autograder directory
2. Run the autograder with pa4.tar as the argument as below:
$python auto_grader.py pa4.tar

12 Grading guidelines

1. We should be able build your program by just running make.
2. Your program should follow the format specified above for the usage interface.
3. Your program should strictly follow the input and output specifications mentioned above. (Note: This is perhaps the most important guideline: failing to follow it might result in you losing all or most of your points for this assignment. Make sure your programs output format is exactly as specified. Any deviation will cause the automated grader to mark your output as incorrect. REQUESTS FOR RE-EVALUATIONS OF PROGRAMS REJECTED DUE TO IMPROPER FORMAT WILL NOT BE ENTERTAINED.)