CS5513 Computer Architecture

Individual Project: Cache Simulator

Due: Dec 7th, 2020, **Monday**, 11:59pm

1 Objectives

In this project, **you will implement a cache simulator that simulates set associative caches with LRU replacement policy**. The goal of this project is to help you understand the internal operations of CPU caches and learn the typical procedure of computer architecture design.

2 General Project Descriptions

2.1 Programming Languages and Reference Systems

You are only allowed to use Python to write the simulator. Python is a very simple language if you know Java or C, and it is an important language to learn. A typical Python implementation of the simulator does not exceed 500 lines of code. If you do not know how to program in Python, this is a good time to start learning it.

Our reference systems for this project are the CS department's fox servers. It can be accessed through ssh to addresses *fox01.cs.utsa.edu*, *fox02.cs.utsa.edu*, ..., *fox06.cs.utsa.edu*. We will grade your submissions on these servers. Therefore, please make sure your simulator can be correctly compiled and executed on these servers. Every student should have an account on these servers. If you have not changed your password before, your user name should be your *abcid*, and your password should be your *banner id* (probably with the "@").

2.2 Simulator Input and Memory Trace Files

Your simulator will need to take one command-line parameter that specifies the the path and name to a memory trace file. This memory trace file includes the memory addresses of a sequence of memory accesses. Your simulator should read in these memory addresses one by one and simulate their accesses to your cache.

More specially, A trace file includes lines in the following format:

```
#PC address:  #R/W #Memory Addresses
0x7f53a8cb3c23: W 0x7fffa80977a8
0x7f53a8cb4860: W 0x7fffa80977a0
0x7f53a8cb4864: W 0x7fffa8097798
... ...
```

The first column of each row is the PC (program counter or instruction address) of the instruction that generates this memory access. The second column indicates whether the access is a read or a write. The last column is the virtual memory address requested by this memory access. All columns are separated with a white space. The size of each memory access is not specified here, although you can assume there are no across-cacheline accesses. Moreover, although real caches typically use physical memory addresses as tags and to compute placement, for this project, you can just use these virtual memory addresses instead.

We provide several synthetic or real memory traces. You can download these traces from here.[1]

Table 1 gives the detailed information for these traces. Note that, the "stride" means the difference in bytes between two memory accesses, not the cache line size.

Trace File	Description	Reference Cache Miss Rate
1KB_64B	Synthetic trace of accessing 1KB memory with 64B stride, repeated for two iterations	50% on 1KB, 16-way cache, 64B blocks/cachelines
4MB_4B	Synthetic trace of accessing 4MB memory with 4B stride, repeated for three iterations	2.08% on 4MB, 16-way cache, 64B blocks/cachelines
32MB_4B	Synthetic trace of accessing 32MB memory with 4B stride, repeated for three iterations	6.25% on 4MB, 16-way cache, 64B blocks/cachelines
bw_mem	Partial memory trace of benchmark *bw_mem*	1.56% on 4MB, 16-way cache 64B blocks/cachelines
ls	Partial memory trace of an *ls* invocation	2.17% on 32KB, 16-way cache 64B blocks/cachelines
gcc	Partial memory trace of a *gcc* compilation	1.89% on 32KB, 16-way cache 64B blocks/cachelines
native_dgemm	Partial memory trace of a naive matrix multiplication of two 256x256 matrices with double-precision floats	50.24% on 256KB, 16-way cache, 64B blocks/cachelines
native_dgemm_full	Full memory trace of a naive matrix multiplication of two 256x256 matrices with double-precision floats	49.37% on 256KB, 16-way cache, 64B blocks/cachelines
openblas_dgemm	Partial memory trace of a cache-optimized matrix multiplication of two 512x512 matrices double-precision floats	8.30% on 256KB, 16-way cache, 64B blocks/cachelines
openblas_dgemm_full	Full memory trace of a cache-optimized matrix multiplication of two 512x512 matrices with double-precision floats	7.5% on 256KB, 16-way cache, 64B blocks/cachelines

Table 1: Memory trace files

The memory trace files are provides as *gzip* files. Please unzip them and use them directly as text files (although Python can read gzip files directly, please still unzip them). Your implementation should have the same cache miss rates as the reference miss rates for these traces (minor differences are acceptable). The reference miss rates are acquired with a popular cache simulator, *CPM$im*.

The real memory traces are obtained with Intel *Pin* and Pintool *pinatrace*. For your experiments, you can download Intel Pin and the pintool to generate traces for other applications. You can also write some synthetic memory traces to test the correctness of your simulator.

2.3 Simulator Internals

Your simulator should be able to simulate set-associative caches of any sizes and any associativities/ways. It is a good idea to make the size and the way of your cache configurable through command-line parameters or configuration files. Note that, broadly speaking, direct mapped cache and fully-associative caches are also set-associative caches. Therefore, your simulator should be able to simulate these two types of cache as well.

For simulation, read in one line of the memory trace file, extract the memory address from the text, and simulate the access with the extract memory address to the cache. If the memory access misses the cache, you can assume that it is served by the memory.

Your simulator should implement the Least-Recently-Use (LRU) cache replacement policy. For the LRU policy, please at least implement a **true LRU** policy instead of the approximation algorithms learned in class. For a true LRU policy, you need to either use a queue or an extra field in the cache frame to record the access time stamp to each cache line. For write policy, please implement the **write-back polity**.

Your cache simulator is only required to simulate the standard operations of a cache. You do not need to simulate every physical function units used in the cache. Additionally, your probably want to read in one line at a time from the input memory trace file, instead of reading the whole files into the memory before processing it.

[1]Full url is,
https://utsacloud-my.sharepoint.com/:f:/g/personal/wei_wang_utsa_edu/EtYXsykz0yxBguZl6K0Q7xsBTwQXzRf98PmurGhGKmihFQ?e=RsBi89

2.4 Simulator Outputs

Your simulator should output the simulation results to the screen (*standard out*, or *stdout*). The output should be a line that reads,

```
Cache miss rate: 66.67%
```

It is important that your simulator outputs only this line, so that we can grade all submissions using scripts. You are free to include other outputs during development phase to assist debugging or experimentation.

3 Project Procedure

In general, your project should carry out in the following steps.

1. **Planning**. Read the slides on Cache Designs and plan your implementation in pseudo-code.
2. **Implementation**. Implement the cache simulator using Python.
3. **Testing**. Test your simulator with different memory traces. First, you should write short memory traces to test different policies of your simulators, including the cache line identification policy, placement policy and replacement policy. After each individual policy is tested, your should test your simulator with large but regular memory traces (such as the provided synthetic memory traces) to ensure the simulator can handle accesses to large sizes of data. In the end, test with real memory traces to ensure your simulator can handle irregular memory accesses.

4 Deliverables and Submission Guidelines

Besides the simulator, please also include a wrapper bash script with the name "*run_sim.sh*". This bash script should accept only one command line parameter, which is the path to a memory trace file. The script should then execute the simulator with the specified memory trace file. The simulator, in turn, simulates the memory accesses in the memory trace and outputs the cache miss rate. We only use this wrapper bash script to grade your simulator. The following commands and outputs show the expected behavior of this wrapper script.

```
$>  ./run_sim.sh a_mem_trace.txt
    Cache miss rate: 66.67%
```

For the submission, please configure your simulator to simulate an LRU write-back cache of 1 MB with 16-ways. If you choose to use Python2, please make sure your bash script uses *python2* as the python interpreter. I recommend every group use python3 with standard packages so that your simulator can be graded properly and simplify the development/submission process.

Please zip your simulator files and the wrapper script into one zip file, and submit the zip file to Blackboard. Please name the zip file "*cache_sim.zip*". There should be no directories in your zip file – all files should be at the root level.

Please strictly follow these submission guidelines. In the cases where the code/script fails to execute, or the files are incorrectly named, or your simulator produces outputs in wrong format, there will be a 20% penalty if we can fix the errors. If we cannot fix the errors, the simulator will be considered as wrong and receives 0 points. All required file names are case-sensitive.

5 Grading

For the simulator, we will execute it with several small memory traces to test if it can produce the correct cache miss rates. The memory trace files used in grading have the exact same format as the provided trace files.