Electrical and Computer Engineering Department, University of Central Florida

---

Spring 2024

EEL 4768 - Computer Architecture Course Project I

Implementing a flexible cache simulator

Due date: March 24th

Contents:

1 Rules and Regulations

1. The work must be done **individually**.
2. For this project, the baseline assessment will be out of **100 points**. Additionally, there's an opportunity to earn up to **20 bonus points** by completing the task detailed in section 7.
3. Any form of cheating and plagiarism such as sharing of your code or using an available code is reported to the school for consequences and results in getting zero.
4. You can communicate with the instructor and the TA through WebCourses regarding your issues.
5. You can use any high-level programming language such as C, C++, java or Python for implementation (preferred C/C++).
6. **Any use of ChatGPT, GitHub Copilot, or such generative AI tools is prohibited**. Never submit content produced by these AI platforms as your own. Committing such an act is plagiarism, which jeopardizes your academic future. UCF's Turnitin tool within Webcourses can detect AI-generated content, providing an "AI score" similar to an "originality score". **Please note**: UCF degree signifies your preparedness for the professional world. While tools like ChatGPT can assist, they aren't substitutes for genuine learning. If opting to use these AI tools, exercise caution and uphold academic integrity.
7. Your project report should be comprehensive and include the demanded results and their corresponding analyses and discussion.
8. Your simulator outputs should nearly match the reference outputs.
9. All your source codes along with the executable file must be delivered.
10. It is recommended to create a "Makefile" or document the compilation command you used.
11. The configurations must be passed to your executable as "arguments".
12. You will receive memory access traces for various applications, complemented by smaller test traces to guide your project's development. Additionally, a skeleton file will be provided to help structure the starting point of your implementation.

2 Project Description

In this project, you need to implement a simple cache simulator that takes as an input the configurations of the cache to simulate, such as: **size, associativity and replacement policy.** When you run you simulator, you need to additionally provide the path of the **trace file** that includes the memory accesses.

Your simulator will interpret the trace file, which adopts the format:

`R 0x2356257`

`W 0x257777`

Each line consists of two parts, the operation type (read or write) and byte address in hexadecimal. After reading each line, the simulator will simulate the impact of that access on the cache state, e.g., the LRU state of the accessed set and the current valid blocks in the set. Your simulator needs to maintain information such as **hits, misses** and other useful statistics throughout the whole run.

In this project, you need to implement two different cache replacement policies: LRU and FIFO. In LRU, the least-recently-used element gets evicted, whereas in FIFO, the element that was inserted the earliest gets evicted.

Implementation hint: allocate your cache as a 2D array, where each row is a set. On each item of the array keep track of information like the tag of the data in this block. You can create multiple instances of such a 2D array for different purposes; for example, you can create another 2D array to track the LRU position of the corresponding block in the LRU stack of the set.

Note: Assume a block size of 64B for all configurations.

3 Inputs to the Simulator

The name of your executable should be SIM, and your simulator should take inputs as following:
`./SIM <CACHE_SIZE> <ASSOC> <REPLACEMENT> <WB> <TRACE_FILE>`
where:

`<CACHE_SIZE>` is the size of the simulated cache in bytes,

`<ASSOC>` is the associativity,

`<REPLACEMENT>` is the replacement policy: 0 means LRU and 1 means FIFO,

`<WB>` is the Write-back policy: 0 means write-through and 1 means write-back and

`<TRACE_FILE>` denotes trace file name with full path.

In general, traces include two files:

- XSBench Application Memory Trace (XSBENCH.t)
- MiniFE Application Memory Trace (MINIFE.t)

Example:

```
./SIM 32768 8 1 1 /home/TRACES/MCF.t
```

This will simulate a 32KB write-back cache with 8-way associativity and FIFO replacement policy. The memory trace will be read from `/home/TRACES/MCF.t`.

Note: the trace file will contain addresses that can be for 64-bit system, so you might need data types that are large enough to read them correctly and bookkeep the metadata in your simulator. For example, if the tag is 9 bytes and you allocate your tag array bookkeeping array as an array of integers, you will not be able to store the whole 9 bytes; integer is only 4 bytes. Accordingly, use data types such as long long int and its equivalents in other languages.

4 Output from the Simulator

The following outputs are expected from your simulator:

a. **Total miss ratio for L1 cache**

b. **The # writes to memory**

c. **The # reads from memory**

It is recommended to ensure that your output formatting is clean, structured, and easy for readers to understand.

5 Testing your code

There are two samples of collected addresses along with the output simulation results found in the provided project files. These resources aim to facilitate verification of the correct functionality of your program, potentially streamlining the debugging process.

The files `smallTest.t` and `mediumTest.t` contain a total of 1,000 and 10,000 address traces, respectively. Both files are bundled in a zip archive named `partialResults.zip`. The commands below can be utilized to verify program correctness:

For the simulation using the file `smallTest.t`, containing 1,000 collected addresses:

```
./SIM 32768 8 0 0 smallTest.t
```

Expected output:

```
Miss ratio 0.143000
write 392
read 143
```

For the simulation using the file `mediumTest.t`, containing 10,000 collected addresses:

```
./SIM 32768 8 0 0 mediumTest.t
```

Expected output:

```
Miss ratio 0.112800
write 2537
read 1128
```

Again, with the file `mediumTest.t`:

```
./SIM 32768 8 0 1 mediumTest.t
```

Expected output:

```
Miss ratio 0.112800
write 34
read 1128
```

And once more with the file `mediumTest.t`:

```
./SIM 32768 8 1 0 mediumTest.t
```

Expected output:

```
Miss ratio 0.117800
write 2537
read 1178
```

Final Validation Runs:

```
./SIM 32768 8 0 1 /home/TRACES/XSBENCH.t
```

Expected output:

```
0.112008
44.000000
2371758.000000
```

And `./SIM 32768 8 0 0 /home/TRACES/XSBENCH.t`

Expected output:

```
0.112008
5013495.000000
2371758.000000
```

6 Grading Policy

1. (20 points) A functional code that runs cleanly, i.e., showing serious efforts to implement the simulator.
2. (40 points) Your code simulates correctly the sample runs we provide and all output statistics should be within less than 0.001% error of our results. 2 tests will be provided along with the configurations and the expected output. Two tests will not be provided to you and are used to make sure your simulator isn't tuned for our sample tests. Each test will weigh 10%.

3. **(40 points) A report that contains the following:**

   A. Use the MiniFE and XSBench provided traces to analyze how the miss ratio of these workloads changes with cache size. Fix cache associativity at 4, write-back, replacement policy to be LRU, and vary the cache size from 8KB to 128KB in multiples of 2, i.e., 8KB, 16KB... 128KB.

   B. Similar to part A, but change write policy and compare between write-back and write-through for each cache size by using the number of memory reads and writes.

   C. Similar to part A, but instead of varying the cache size change the associativity. Fix the replacement policy to be LRU, cache size to be 32KB, associativity to change from 1 to 64, in multiples of 2, e.g., 1, 2, 4... 64.

   D. Similar to part C, but now study the impact of replacement policy. Specifically, run the same experiments you ran in part B, but with FIFO replacement policy, write-back, then compare the results of part C and D to study the impact of replacement policy changes with associativity.

   In each part of your report, plot the results and explain the reasons behind the trend and compare the trends of MiniFE with XSBench.

7 Bonus Opportunity

As an advanced challenge, you are invited to implement one or more of the following cache replacement policies. **Successful implementation can earn you a bonus of up to 20 points**. It's essential to highlight that for this bonus task, no test outputs will be provided. The bonus points will be awarded only if your implementation's output aligns within a 5 percent error margin of the expected result.

- *Last-in-first-out (LIFO) policy:* The most recently inserted item in the cache is evicted first.
- *Most-recently used (MRU) policy:* The most recently accessed item is evicted first. This approach is beneficial when the likelihood of the same request repeating in the near future is low.
- *Least-frequently-used (LFU) policy:* This policy maintains a count of the number of times a cache item is accessed. The least frequently accessed item is evicted to accommodate new items.

Please be aware that not all these methods may have been covered during our class sessions. Exploring beyond the curriculum embodies the spirit of this bonus challenge!

Best of luck!