Programming Assignment 1 (Catch the CACHE)
CS 773: Computer Architecture for Performance and Security, Fall 2022
Computer Science and Engineering
Indian Institute of Technology, Bombay
Due Date: August 16, 2022, 6 PM IST
Submission Link: Moodle (https://moodle.iitb.ac.in/login/index.php)

This programming assignment is designed to help you get a good understanding of the basics of cache and **cache hierarchy**, to get your hands dirty performing **experiments** and to get familiar with the world of microarchitectural tools and **simulators**. This assignment is an individual assignment.

Task 1: Super easy (10 Points)

After attending Biswa's lecture on caches, Catch got confused about the size of the caches on his/her laptop. Tried various commands and got ad hoc numbers :(After a while, Catch decided to do it his/her way. As a friend of Catch, measure the **size of each level of cache** of your system. Most of the modern systems have three levels of cache (L1 to L3) with sizes ranging from few KBs to MBs.

Recommended method:

- Time the accesses for an array.
- Vary the size of the array and record the corresponding access times and plot the array size vs execution time.
- Assuming you have done everything correctly till this point, you will see certain jumps in access times in your plot. You need to conclude the cache size of each cache level by analyzing this plot.
- You can validate your results by using **perf** tool to view the actual number of cache misses at different array sizes.

Points to note:

- You are not allowed to use ready-to-use commands like **lscpu or getconf** rather you need to ***measure*** the cache size. Although you need to compare your measured values with the actual values (using the above commands).
- Do keep in mind the effect of prefetcher on your experiments, the reason being prefetchers will fetch the required data ahead of time in the cache which will nullify the compulsory misses required to correctly identify the cache sizes.
  For example, to disable the prefetcher for all the cores on AMD 10H family execute the following commands:

```
$ sudo apt-get install msr-tools
$ sudo modprobe msr
$ sudo wrmsr -a 0xc0011022 0x2000
$ sudo rdmsr -a -x -0 0xc0011022
```

You need to figure out the appropriate settings to disable the prefetcher depending on your processor family.

Deliverables:

1. Source code along with proper documentation. Name the source file as cache-code-task1.c
2. Scripts for running code and generating plots. Put them in the task1/code/ directory.
3. Report should contain the following:
    - Thorough explanation of your approach
    - Measured vs actual cache size of each level. The actual cache sizes can be found using **lscpu** command.
    - Plots along with a brief description of how you derive the measurements from the plots.(max. 30 words)

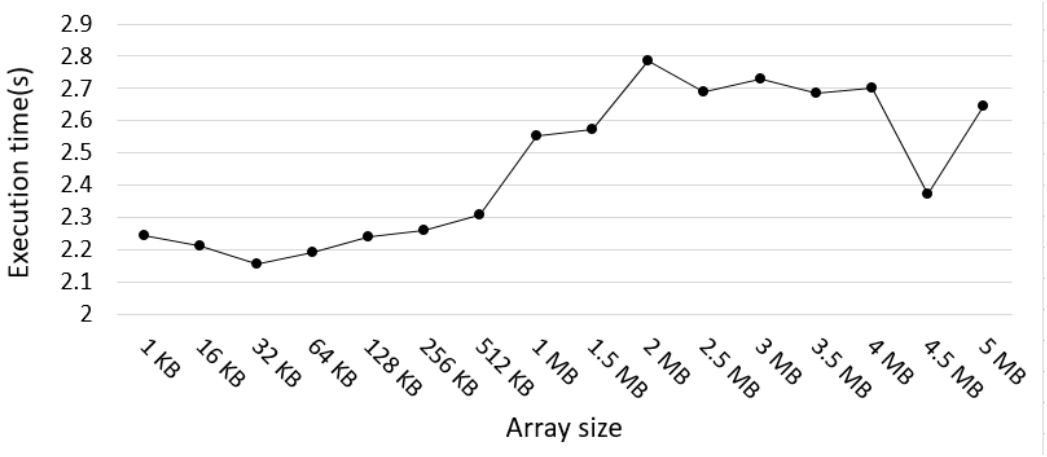

Fig: A sample plot showing the change in execution time for different array sizes.

Refer to the submission format section for the directory structure

Task 2: Welcome to the world of simulators (10 Points):

This part of the assignment will also deal with a similar goal, but through a micro-architectural simulator called **ChampSim**. Please use this version of Champsim for this task. Your goal is to determine the size of L1D, L2C, and the last-level cache(LLC, a.k.a. L3C). This can be done by using variations of a program that accesses an array of a specific size, say 16KB, 32KB, or 64KB, and then observing the instruction per cycle (IPC) difference. If at a specific array size,

there is an IPC drop, it means the previous array size was the size of the cache. Going from 1KB to 8MB, you will first get the size of L1D, then L2, and finally LLC.

Recommended method:-

1. Put your code in `cache-code-task2.c`. The code should take array size as a command line arg.
2. Generate traces using `cache-code-task2.c` to access different array sizes (Example: You can use log scale for input i.e $\log_2$ of the array size to be accessed. For example, for 4KB, input should be 12; for 8KB, input should be 13, and so on). **Generate traces to access array sizes starting from 1KB to 8MB with different names** (use `-o` parameter with the Pin-tool to specify the output trace name). Skip the first 10 million instructions and generate traces for the next 2 million instructions. Use `-ifeellucky` parameter with the Pin-tool.
   Compress the trace files to `.gz` or `.xz` format
3. Build ChampSim with `bimodal` branch-predictor, no prefetchers, LRU replacement policy and single-core processor
4. Run the generated traces with a warm-up of one million and simulation of one million instructions
5. Note the difference in IPC, for different sizes of the array and verify the results by checking with the actual cache size settings in the champsim. The cache size can be computed using number of sets, number of ways, block size. The values can be found in `inc/cache.h` and `inc/champsim.h`

Deliverables:

1. Source code along with proper documentation. Name the source file as cache-code-task2.c
2. Generated trace files for every array size. The name of the trace file should be `x.champsimtrace.xz`. Here x is $\log_2$(array size). (For example, if array size is 4KB then x is $\log_2$(4KB) = 12)
3. Output files generated by champsim by running with the generated traces. The name of the output files(from champsim) should be `x.txt where` x is $\log_2$(array size).
4. Scripts for running code and generating plots and put them in the task2/code directory.
5. The report should contain the following:
   a. Thorough explanation of your approach
   b. Measured vs actual cache size of each cache level in the champsim.
   c. Plot showing array size vs IPC along with a brief description of how you derive the measurements from the plots. A sample plot is shown in the figure below.

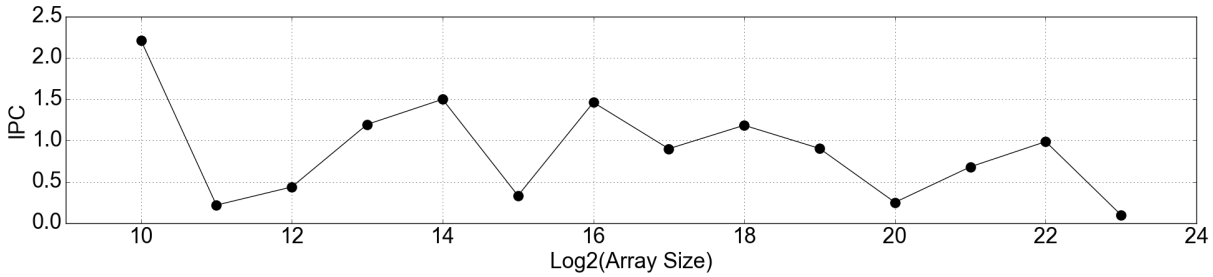

Fig: A sample plot showing the change in IPC on different array sizes.

Refer to the submission format section for the directory structure

Bonus Task(5 Points):

Measure the **cache block size** of your system.

Hints:

- Access an array with varying strides(1, 2, 4, …) for a fixed number of times.
- Record access times for each stride.
- Strides that cross the cache block boundary will lead to a sharp jump in access time.

Deliverables:

1. Keep your code files in the `bonus/code/` folder.
2. Include the step-by-step instructions on how to run your code in `bonus/howtorun.txt`
3. Thorough explanation of your approach(`bonus/report.pdf`).

Refer to the submission format section for the directory structure

Submission format:

1. You should submit a single tar.gz file with the name `roll_number.tar.gz.`
2. The folder structure within the tar.gz should be in the below format. Place the files in appropriate folders for all the tasks. You may clone this github repo to get the proper directory structure.

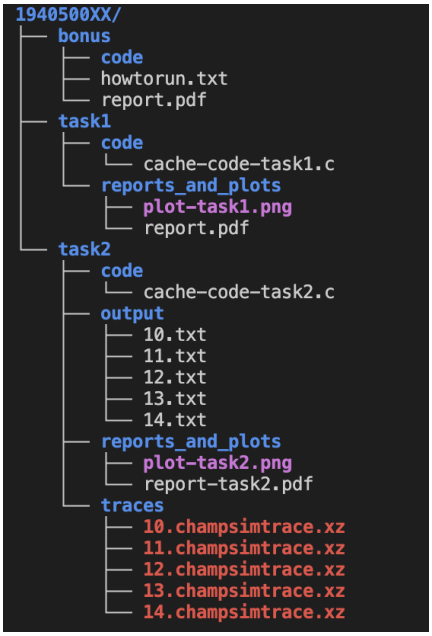

Fig: Sample directory structure of the submission. Note that the output files(for task 2) should range from `10.txt` to `23.txt` in your submission (same for traces). For the sake of clarity, only 5 files are shown above in the corresponding directories.

Appendix

Preparing a Trace

In order to evaluate the performance of a system on a specific benchmark, with ChampSim, we are required to prepare a trace-file for that specific benchmark. Trace-files, or simply traces, are special files that contain information about run-time behavior of a program - the addresses to which loads/stores were issued, the number of instructions executed etc. As for the details on how to write a program to track relevant things and put them inside the trace, is beyond the scope of this course (and is not at all required to work with ChampSim, since it is already provided) but what is required is to be *able to use it*.

In the following steps below, a **hypothetical program** `hello_world.cpp` will be used as the example from which a trace file will be generated.

1. Clone ChampSim repository from here
2. Download the appropriate version of Pin depending on the version of the Linux kernel in your system (Refer to the ChampSim repository's "How to create traces" section, for more details)

3. Enter `tracer/` directory inside ChampSim and open `make_tracer.sh` in your text-editor. Change the value of `PIN_ROOT` to the location where you downloaded (and extracted) Pin
4. Execute `make_tracer.sh` to build the tracer for ChampSim
5. If you do everything correctly, the build will finish successfully and a directory `obj-intel64/` will be created with two files inside it: `champsim_tracer.*`
6. Compile `hello_world.cpp` normally to get the executable, say, `hello_world.out`
7. Change to Pin's directory and execute the following command (in a single-line)

   ```
   ./pin -t PATH_TO_CHAMPSIM/tracer/obj-intel64/champsim_tracer.so
   -o hello_world.champsimtrace -t N_TRACE -
   PATH_TO_EXECUTABLE/hello_world.out
   ```

   where
   (a) `PATH_TO_CHAMPSIM` needs to be set to the location where you have ChampSim repository cloned
   (b) `N_TRACE` needs to be set to the number of instructions to trace (for example, 1000000)
   (c) `PATH_TO_EXECUTABLE` needs to be set to the location where you have your executable in
8. If you do everything correctly, after few seconds (or minutes), you will have a new file named hello world.trace generated by Pin.
   If there is some issue (due to kernel version mismatch with Pin), try using `-ifeellucky` flag. If the issue persists, ping us.
9. The final step is to compress the generated trace using `xz` utility. While being in the same directory (where the trace was generated), execute the following command

   ```
   xz hello_world.champsimtrace
   ```

10. Finally, you will get a file named `hello_world.champsimtrace.xz`, which is the trace that you will be using to evaluate the performance of a system using ChampSim

Late submission policy:

- As per lecture-1 logistics.