Mission 1: Exploring Cache Architecture and Access Timing

Computer Architecture & Hardware security

1 Mission Briefing:

No cool gadgets (unless you count your keyboard), no disguises (unless you're hiding Linux under a Windows hoodie), and no fancy AI assistant whispering in your ear. All you've got is your brain, some C code, and a processor that leaks more secrets than your college WhatsApp group during exam season.

Your job? Figure out what your machine is really made of, cache sizes, associativity, number of sets and what not. Don't worry, no lasers, no explosions. Just some serious cache digging and a few well-timed CLFLUSHes.

Part 1.1: Determining Machine Architecture

To understand your machine's cache hierarchy, use public system interfaces on Linux. Run the following commands:

- `lscpu`: Shows processor type and general architecture info.
- `less /proc/cpuinfo`: Detailed info for each logical processor. (Press `q` to exit.)
- `getconf -a | grep CACHE`: Provides cache configuration (sizes, line size, levels). Units are in **bytes**.

For more in-depth architecture specs and latency numbers, refer to WikiChip: `https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake`

Task

Complete the following table using the gathered system information:

Cache	Cache Line Size	Total Size	Associativity (Ways)	Number of Sets
L1-Data				
L2				
L3				

Use the following formula to compute the number of sets:

$$\text{Number of Sets} = \frac{\text{Cache Size}}{\text{Line Size} \times \text{Associativity}}$$

Part 1.2: Measuring Memory Access Latencies

In this part, you will measure the latency (in CPU cycles) to access data from two different levels of the memory hierarchy:

- Cache (could be L1, L2, or L3, any data that hits in cache)
- DRAM (data that is definitely not in the cache)

Objective

Your task is to write your own C code to measure and report:

- Average latency to access data from the cache
- Average latency to access data from DRAM

Helpful Gadgets

- Use **rdtsc** to capture the timestamp before and after memory access.
- Drop an **lfence** in between, not a medieval weapon, but it'll keep things in order.
- Use **clflush** to push data out of cache, think of it as clearing the crime scene before the next operation.
- Measure each case multiple times. Cache is quick, but also sneaky, it might lie once or twice.

You are expected to write the code from scratch. Use your knowledge of C and low-level programming to directly interact with memory and CPU instructions.

Task

Present your measured values in the following table:

Memory Type	**Measured Latency (cycles)**
Cache	
DRAM	

Briefly explain your method and show a Plot to support your result. A sample plot is given below. Also submit your code.

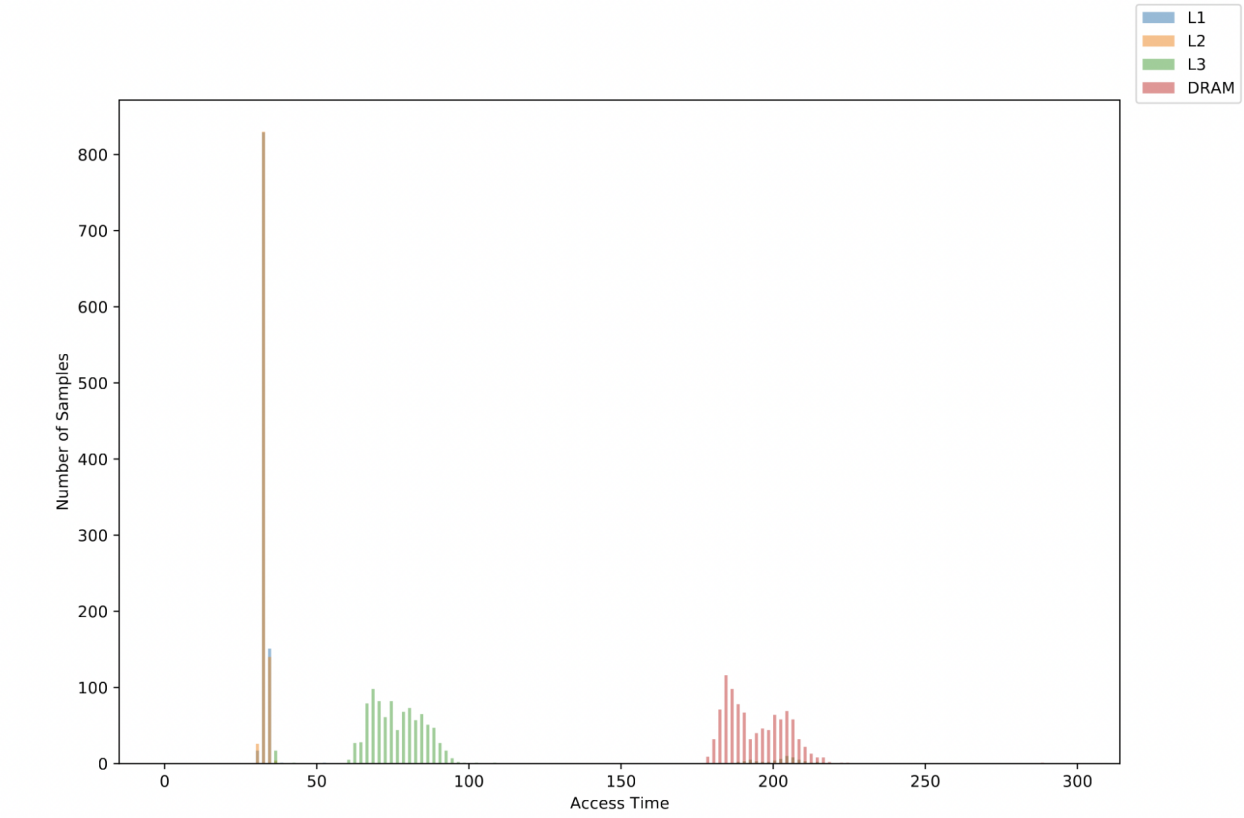

Figure 1: Acess latencies for measured multiple times (Samples)

Good luck for the mission!