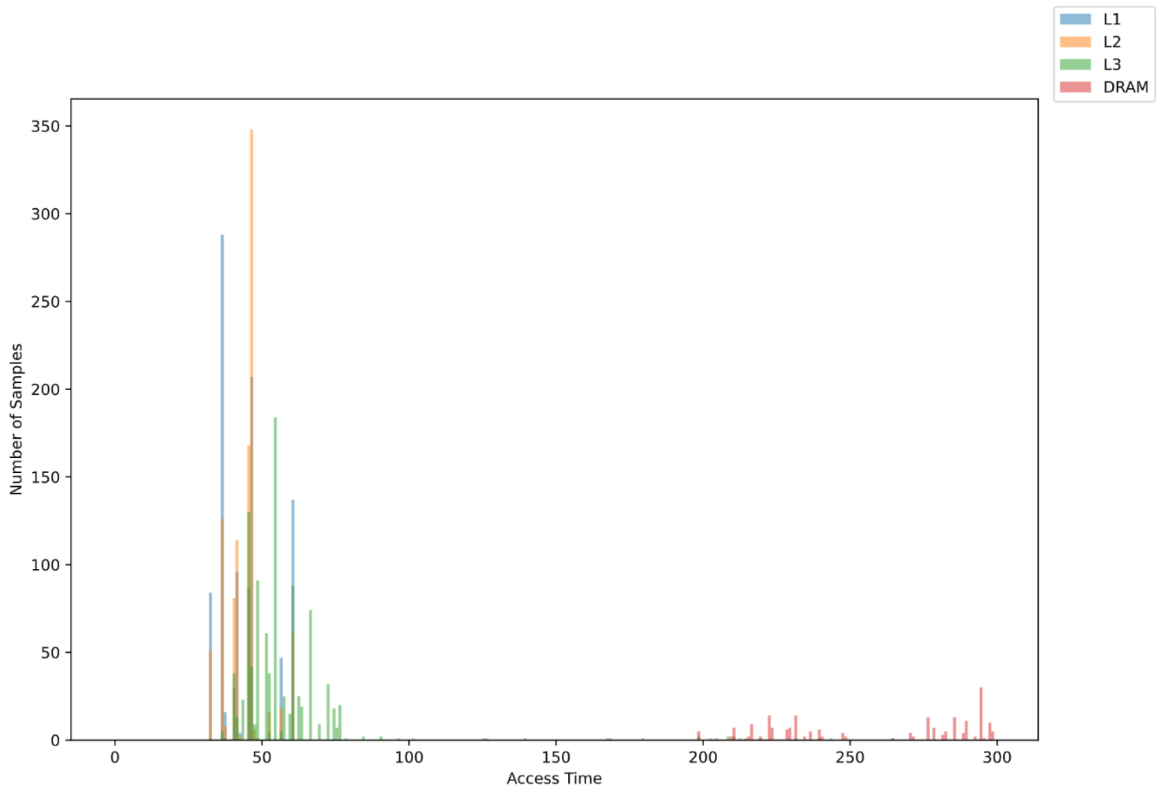
Part 1: Reverse-engineering cache and memory latencies.



A clear distinction between L1, L2, L3 and DRAM can be seen from this experiment.

Discussion Ques 4: Describe your communication protocol to communicate a single bit value.

I am creating a L2-cache covert channel between the sender and the receiver through which they can communicate a single bit value with high reliability. The communication protocol is described below:

<u>Sender</u>

1. Asks for a memory buffer having the size 1.5 times of L2 (1.5*256 KB).
2. Samples user input continuously.
    a. If input is ‘1’, it accesses each line of L2 using the memory buffer. Prints ‘Sent 1’ in console.
    b. If input is anything else, it does nothing.

<u>Receiver</u>

1. Asks for a memory buffer having the size of L2 (256 KB).
2. Accesses all the L2 cache lines using this buffer.
3. Again, accesses all the lines 10,000 times, now measuring the cache access time for each line. Calculates a baseline time to access the whole buffer.
4. Listens continuously.
    a. Measures the access time for each cache line in a non-linear fashion to trick prefetchers.
        i. If this time is greater than the baseline time + delta, a miss is recorded.
        ii. Else, hit is recorded.
    b. After each pass through the L2 cache, miss ratio (= #miss / (#miss + #hit)) is calculated. If this miss ratio is greater than a pre-defined threshold, a ‘1’ is expected to have been sent by the sender. Prints this message in console.

<u>Reliability of this communication protocol:</u>

In my experiments, the average true positive rate was ≈70% and false negative rate was ≈20%. Although, this greatly varies depending upon the two threshold parameters and noise in the system.

Exercise 6: Derive the threshold for the decode operation. Implement a 1-bit chatting client.

The decode threshold comes out to be 37 cycles and miss ratio of 0.26. The chatting client has been implemented.

Discussion Ques 7: Describe your communication protocol.

The 8-bit communication protocol is quite different from the single-bit protocol. Here, I took advantage of hugepage to create eviction buffers on both sender and receiver side that map to the same set indexes. The protocol is described below:

<u>Sender</u>

1. Requests for a hugepage of size 2 MB. Asks for a memory buffer having the size of L2 (256 KB).
2. Iterates through the memory buffer, looking for the line having all set bits = 0. Captures that address in another pointer ‘eviction_buffer’.
3. Samples user input continuously.
    a. Converts the string into a binary string. The lower indexes of this binary correspond to the actual message, and the upper indexes are the newline character.
    b. For every character in binary string, if it is a ‘1’ all the lines of 8-way set-associative cache are loaded with the corresponding eviction set. If ‘0’, nothing is done.
    c. An additional set #9 is also accessed, it is used by receiver to determine whether something is sent from the sender side.

<u>Receiver</u>

1. Asks for a memory buffer having the size of L2 (256 KB).
2. Finds an eviction buffer like sender, starting with set index 0.
3. Finds the baseline hit time for this buffer through the same approach described for 1-bit.
4. Listens continuously.
    a. For each set (index 0-15), it measures the access time for all the 8 cache lines. If it is greater than baseline + delta, a miss is recorded for that line. If there are more than 4 misses out of 8 for a set, a ‘1’ bit is recorded corresponding to that set in the message string.
    b. If more than a threshold number of misses are detected in set #9 (meaning something was actually sent by the sender), the message string is converted to ascii value. The result is displayed in console.