COMPX304 Assignment 1: Side Channel Attacks

Stefenie Pickston

08-04-2022

Abstract

Side channel attacks are a non malicious method for gaining information about a computer system. In this report, a side channel attack will be used in order to find the size of the Last Level Cache (LLC) in the CPU.

A method to find information about the CPU cache level sizes is to use an array and access different portions of it using a loop. This is because the run time will be determined by accesses in the computer's memory. Read/write accesses inside the cache and outside of the cache will have timing differences and this can be measured in order to make a comparison.

This report will cover the software design of a side channel attack, the tests conducted and discussion of the test results.

1 Software Design

1.1 Introduction and Preliminary Tests

I am working from my laptop which runs Ubuntu natively and has an Intel i5 6200U CPU. According to lscpu my laptop CPU has cache sizes of:

Cache Level	Size	Total Size
L1	128 KiB	128 KiB
L2	512 KiB	640 KiB
L3	3 MiB	3712 KiB

Note that the L1 cache is made up of both the L1d and L1i which are 64 KiB each.

As a preliminary test, I wanted to test the step values for the lines in the cache. I used code from Igor Ostrovsky's blog to gauge an idea of how I could measure this.

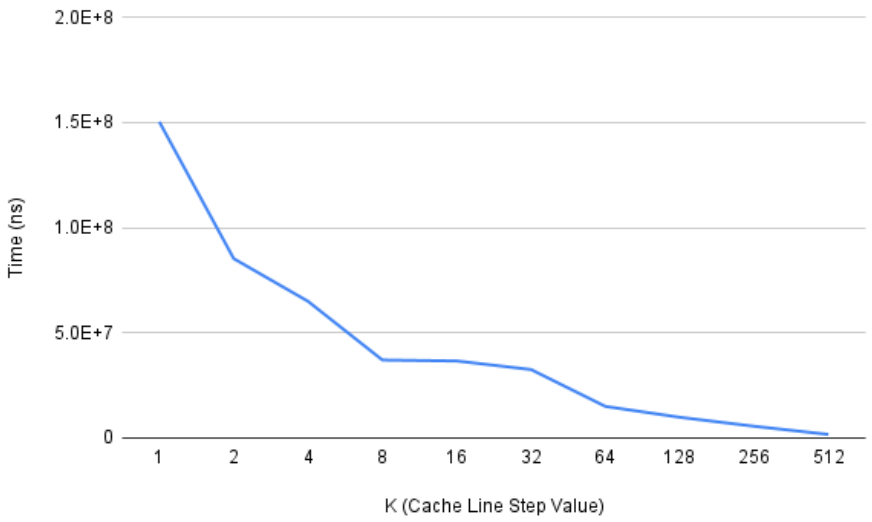

Figure 1: Cache Line Step Values

From the graph, each time K (the cache line step) doubles, the run time of the procedure halves. CPU's tend to fetch memory in cache lines instead of byte by byte. However from the data it looks like that the CPU recognizes the pattern and the prefetch has improved in comparison to older CPU's where it would fetch the next block of data instead.

Therefore in order to test for the cache sizes, any K equal or over the value of 2^5 will be sufficient. I will be using K = 64 for my tests as this is the cache line size of my CPU.

1.2 Testing the L1 and L2 Cache

Using Igor Ostrovsky's code example as a starting point, I ran tests to measure the size of the L1 and L2 cache. These tests measure each cache hit individually. I modified the code to suit my approximate cache sizes, minimise noise as well as allow data collection.

```
for(int i = 8; i < 1024*X; i += Y){
    System.gc(); // garbage collector
    long total = test(i); // run test
    System.out.println(total + "," + Math.floorDiv(i, 1000));
}
```

Where X is a rounded number above the cache size that we are looking for in KiB and Y is a power of 2 which modifies the array length to be tested which will allow easier aggregation of the results. I also modified the code to use a byte array for easier calculations.

```
byte[] arr = new byte[size];
int steps = 64 * 1024 * 1024; // arbitrary number of steps
int lengthMod = arr.length - 1;
for (int i = 0; i < steps; i += 64){ // i += cache line size
    arr[(i * 64) & lengthMod]++;
```

There will be peaks (time delays) whenever a higher level of cache has been hit. This is because it has to suddenly stop and pull data from main memory or the next level.

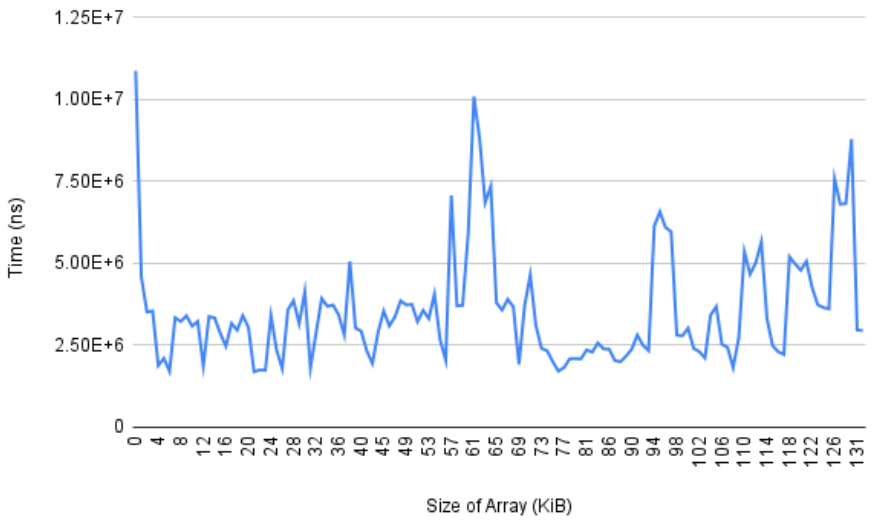

Figure 2: L1 Cache

The L1 cache has peaks at 61 KiB. The 61 KiB peak correlates to the size of the L1d cache where performance is affected.

I had to spend more time on aggregating the input to get clearer data for future graphs, since my laptop has lower specs and generated a lot of noise in the data.

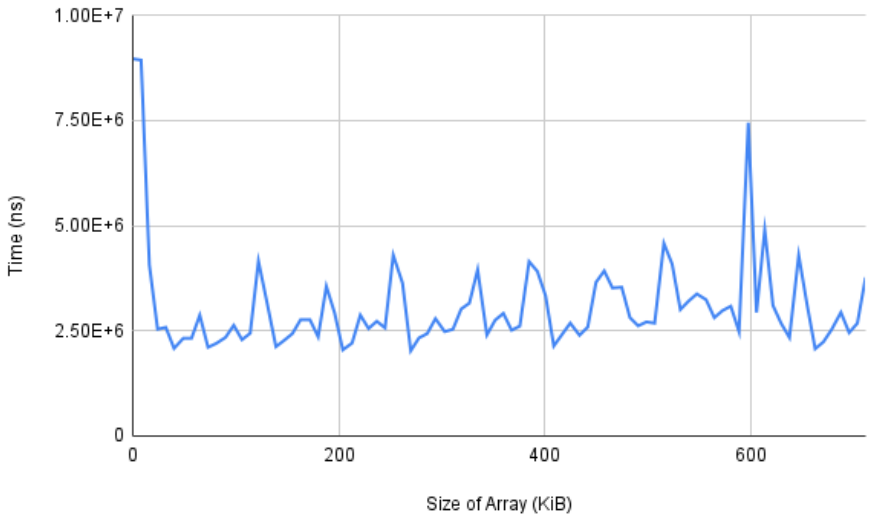

Figure 3: L2 Cache

In the L2 cache there is a peak at 598 KiB, despite lscpu specifying that the total L1 and L2 cache size should be 640 KiB. This is most likely due to the 598 KiB being used by the array and the remainder 42 KiB being used by the OS or other background processes. The cache realizes that the array process is linear so it will evict items back in the cache in order to free up space to finish processing the array. Then it goes back to fast access for the last portion of the array.

We can infer the size of the L2 cache because there will be some background processes, which will affect when the spike occurs.

2 Impact of the Array Size

As the array gets bigger in size, there are more items to keep in the cache. Thus it will use more cache space and the cache size will grow. When all the cache levels (L1, L2 and L3) are full, this spills over into main memory (RAM). I decided to do a test to show this happening by timing the entire iteration.

```
for(int i = 1; i < 17; i ++){
    System.gc(); // garbage collector
    long total = test((int)Math.pow(2, i) * 1024); // run test
    System.out.println(total + "," + (int)Math.pow(2, i));
}
```

I modified the previous test's code to run slightly differently where I changed the size of input per loop iteration, which would yield broader output.

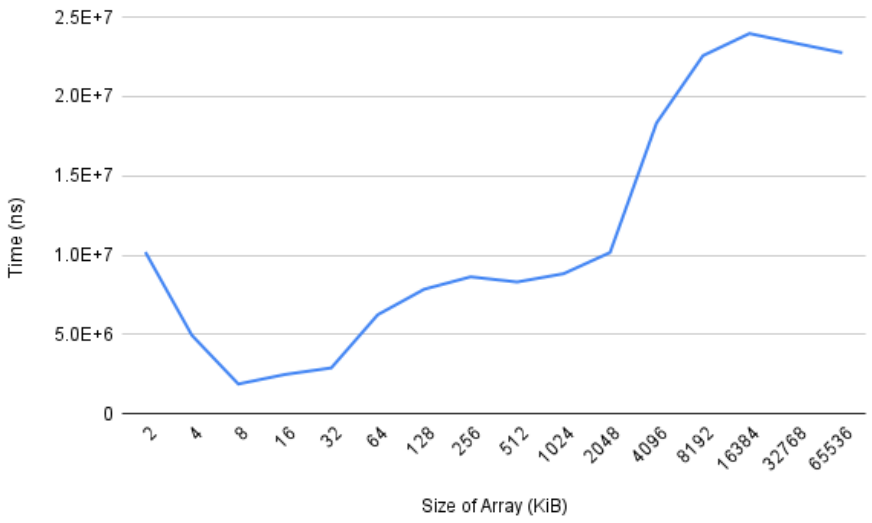

Figure 4: Logarithmic Cache Measurements

From the logarithmic graph we can conclude that the array filling up does go into memory. There is a larger jump between 2048 KiB and 4096 KiB suggesting that the L3 cache to memory boundary is within this range.

3 Finding the Size of the L3 (Last Level) Cache

With an understanding of how the array size affects the cache, I ran some tests to calculate the size of the L3 Cache and when it jumped into main memory. These tests measure each cache hit individually, similar to the tests written for the L1 and L2 cache sizes.

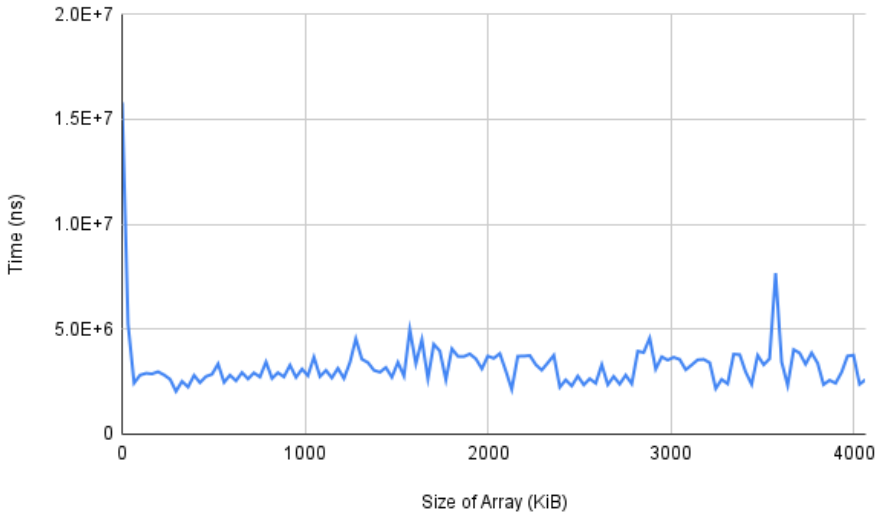

Figure 5: L3 Cache

There is a spike at 3571 KiB, and this shows the point that the process jumps into main memory. Again, this value is smaller than lscpu cache total size of 3712 KiB, as the remainder 141 KiB is being used by the OS and background processes. Therefore we can infer the total size of the cache which is 3712 KiB, and the size of the L3 cache itself which is 3 MiB.

In conclusion, the tests conducted have shown the cache sizes with a reasonable amount of accuracy. However, this was only because I made sure there were as few background processes running as possible. In order for more accurate testing of the CPU, similar tests would have to be conducted where the CPU runs nothing else but the test.

Ostrovsky's blog:
`https://igoro.com/archive/gallery-of-processor-cache-effects/`