Side Channel Analysis of Last Level Cache(LLC) Size in Java

Ryan Unternahrer

May 28, 2024

1 Introduction

The central processing unit (CPU) of a computer uses a set of caches to store data that may be needed soon closer to the CPU. This removes some of the need to read from the random access memory (RAM) and therefore increase the speed of program execution. Our goal is to be able to infer the size of this cache without direct access to that information. This is called a Side Channel attack and relies on indirect methods to gather information about, or to influence the execution of, a piece of software.

Most modern computers have a three level system of caches[1] with the Level 1 (L1) cache being the closest to the CPU, and therefore quickest to access, and the Level 3 (L3) cache being the furthest from the CPU and the slowest to access. If a piece of data is not stored in any of the caches and is instead in the main memory when a program attempts to access it a cache miss occurs. This will often result in a significantly slower speed to access this data.

2 The Tools

2.1 Java

All code for this project will to written in the Java programming language. We will rely on the `System.nanoTime()` command to measure time. This returns data in Nano Seconds (ns).

2.2 Linux Environment

We are required to write and test all code in a Linux development environment. To fulfill this requirement I will use a Windows Subsystem for Linux

[1] $WikimediaFoundation.https://en.wikipedia.org/wiki/CPU_cache$

(WSL) running Ubuntu. This allows us to compile and run code as though we were using a Linux machine.

3 The CPU

For this assignment we will use my laptop to run all code therefore it is the cache of my laptop that we will be measuring. Below are the stats of my laptops CPU.

CPU Info	
CPU Name	Intel Core i5-10300H
Clock Speed	2.50GHz
L2 Cache Size	1024 kilobytes
L3 Cache Size	8192 kilobytes

Table 1: Laptop CPU Info

As can be seen here there is a Level 3 (L3) cache of 8192 kilobytes on my CPU. We want to overflow this cache in order to induce a cache miss. This means we should expect to see a spike in access time around 8MB in array size.

4 Theory

We will attempt to intentionally induce a cache miss. Our method for this will be by creating an array that is too large for the system to hold all of its data in the CPU cache. We want to measure many different sizes so we can create a trend in the data and look for a spike in the amount of time to access this information. To avoid any outlier cases we will run each size of array 100 times and average the data. Therefore the output result for each array size will be: $average = \frac{totaltime}{100}$.

Our first method will be to randomly increment values in the array so that we do not create any kind of pattern to our access. Once again to avoid outlier cases we perform this a large number of times. For the purposes of this experiment we will increment values 1,000,000 times in each run. This means for each size the speed of access will be measured 100,000,000 times which should be a sufficiently large dataset. I will time and average each run rather than each access of data as the time to access data may be too small to measure especially for data store within the cache.

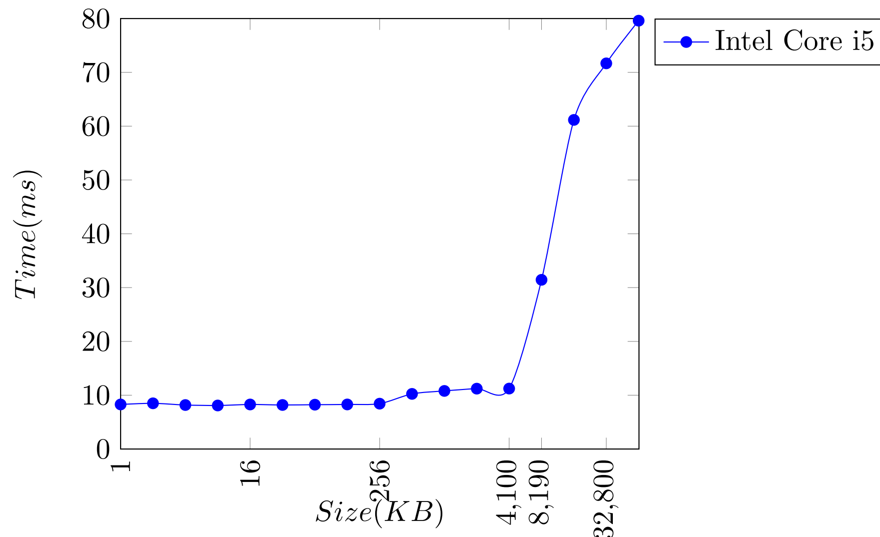
Our second method will be to sequentially access the data in the each array.

This is an attempt to see if the accessing the data sequentially effects our ability to infer information about the size of the LLC. To avoid the varying sizes of array effecting our data if an array is too small for 1,000,000 separate entries then the program will loop back to the beginning of the array.

5 Random Access

For the testing of random access I wrote the program `cachetimerandom.java`. I based my program on ideas and code in the provide article from igoro.com.[2] This program tests arrays from a size of 1 Kilobyte (KB) to 64 Megabytes (MB) with the size doubling each time for a total of 16 sizes. Each test consists of 1,000,000 increments of the data.

Average Time to Increment 1 Million Values in Arrays from 1KB to 64MB



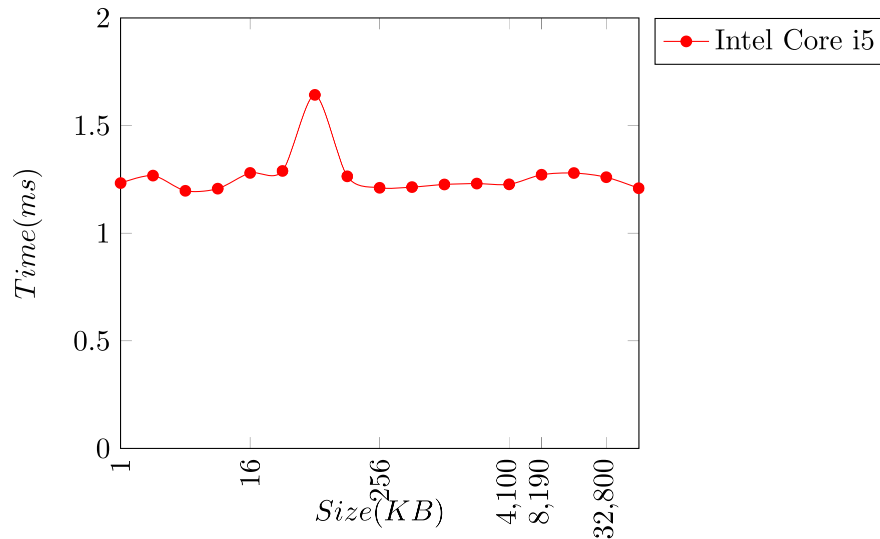
As can be seen in the above graph the time to increment 1,000,000 values in an array is roughly 8-10ms for arrays between 1KB and 4MB. As the array size reaches 8MB the time required to perform the increments increases to roughly 35ms and continues to climb. From this data we can infer that the size of the LLC on the computer is roughly 8MB. This falls in line with what we expect as in Table: 1 the CPU being tested has a LLC size of 8MB. This represents a successful side channel attack as we were able to infer information about the computer without having direct access to that information.

[2]Gallery of Cache Effects https://igoro.com/archive/gallery-of-processor-cache-effects/

6 Sequential Access

For the testing of random access I made a slight modification to my program that allows for the elements of an array to be incremented sequentially rather than randomly as in the previous test. This is done by tracking variable that increments each time it is used and resets to zero when it reaches the maximum size of the array. This program can be found in `cachetimeseq.java`.

Average Time to Increment 1 Million Values in Arrays from 1KB to 64MB

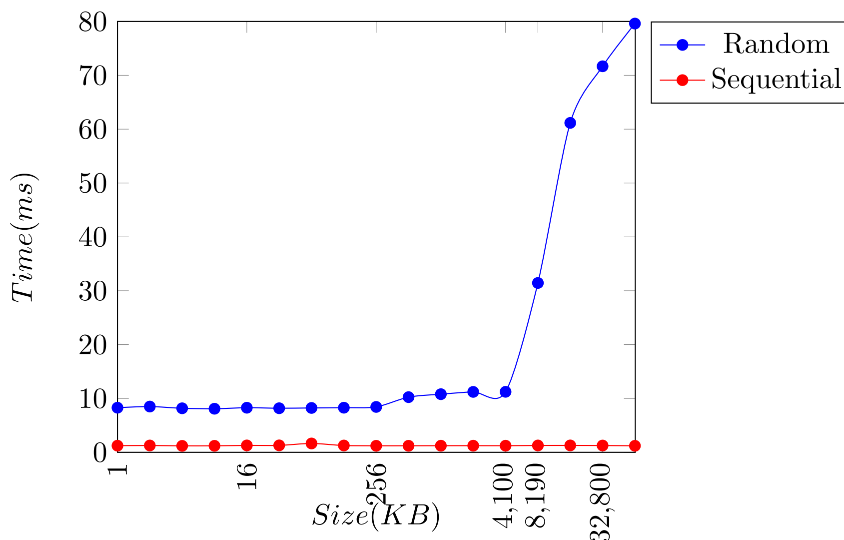


As we can see in the above graph the the values tend to stay around 1.2ms per run of 1,000,000 increments. There is a small spike at 64KB but this is an outlier and though we ran each size 100 times to try avoid such outliers they are still possible. We can see here that there is no large spike in time with the sequential access method.

7 Comparison

In this section we will compare the two methods and discuss how they differ. Below are the two methods plotted together.

Average Time to Increment 1 Million Values in Arrays from 1KB to 64MB



As can be seen above both methods start with a relatively flat trend this is due to the array being fully cached up to a value of 8MB. After that we can see the random method spikes in time as the array cannot be fully stored in cache. The sequential method however remains on a flat trend throughout the test. The sequential method also starts much lower than the random method. I believe this is due to some form of predictive caching that retrieves information for the cache in sequence as the order in which information is needed can be predicted unlike the random method. This makes a sequential method of iteration unusable for a side channel attack on this processor and others that use a similar technology.

8 Conclusion

We have tested two different methods of performing a side channel attack to infer the size of the LLC on a processor. The use of a random access method showed very effective results and allowed us to accurately infer the size of a CPU's LLC. We also found that a sequential method of access was not suited to performing side channel attacks as this method did not give us any information that could be used to infer the size of the LLC. Therefore we can determine that a random method of accessing data is much more effective at performing side channel attacks in relation to cache size.