Side-Channel Attacks and the LLC

HK Transfield

Introduction

Within a Central Processing Unit (CPU), is a fast memory unit known as a memory cache.[1] Considering how close this cache memory is to the processor, it is much smaller than the main memory; therefore, it has less storage and is not expensive. Due to the speed and cost of using cache memory, in modern CPUs the cache is divided into three levels, describing how close and accessible it is to the CPU. The highest of those levels, L3, is the LLC (Last Level Cache), which is shared by the CPU core. The size of the LLC is larger than the other levels that were developed to improve the performance of L1 and L2, hence why L3 is slower than the higher levels.[2]

The size of the LLC can be verified through timing read/writes of large arrays. This type of operation utilizes how certain side-channel attacks are performed, where the attacker will probe a victim's cache and observe the execution time, revealing which cache line the victim has accessed through measuring the time difference.[3] Therefore, the purpose of this report is to explain the design and testing of a simple program that can infer the size of the LLC.

1.0 Program Design and Algorithm

The program used to infer the size was written in Java and was inspired by the algorithm designed by Igor Ostrosky.[4] The algorithm loops over a large array, incrementing every 64th byte so that every cache line is easily modified.[a] Any time a memory location is read or written within that 64-byte region the entire cache line is fetched from the main memory into the cache.[5] If it were to reach the final indices in the array, the loop will go back to the beginning of the array. This loop will be referred to as the inner loop of the program.

[a] A cache line is a 64-byte unit of data transferred between the cache and main memory

```
lengthMod = arr.length - 1;
for (int s = 0; s < steps; s++) {
      arr[(s * 64) & lengthMod]++;
}
```

Using this algorithm, we can experiment with measuring performance by looping through different array sizes which exponentially increase from 1 kB to 256 MB with each index, covering the size of most modern CPUs — this will be the outer loop of the program.

Since Arrays have a constant execution time when accessing a single item, written as O(1) in Big O notation, the inner loop should execute in a constant time too.[6] The expectation is that as the Array grows larger, the time to read from them should also increase, with distinct drops for every cache level passed. For every array size, the program runs it up to 50 loops and averages the result.

2.0 Testing Results

2.1 Multiple Machines

The program was tested across multiple different machines to verify that it can measure the LLC. The expectation is that the relationship between array size and access time will result in a curve appearing as a staircase indicating that there is a jump from one cache to the next level.[7] When the array becomes larger than the LLC and needs to read from the main memory, there should be a noticeable spike in access time, which can be used to infer what the LLC is.

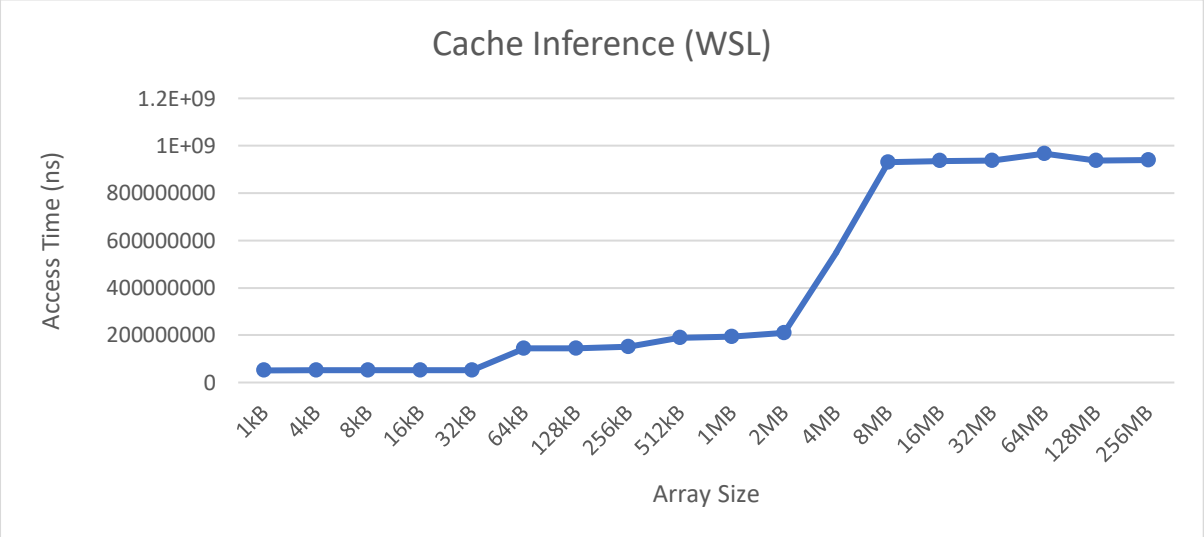
2.1.1 Windows Linux Subsystem

The first machine used in testing was an HP ZBook 17 using WSL (Windows Linux Subsystem), to test the program on an Ubuntu environment directly on a Windows 10 machine.[b]

[b] Refer to the Microsoft docs on WSL. https://docs.microsoft.com/en-us/windows/wsl/about

HP ZBook 17 Specifications	
Architecture	x86_64
Address Sizes	32-bit, 64-bit
CPU(s)	4
Model name	Intel(R) Core(TM) i5-4300M CPU @ 2.60GHz
L1d cache	64 KiB
L1i cache	64 KiB
L2 cache	512 KiB
L3 cache	3 MiB

There are distinct plateaus and small jumps from 64kB and 512kB, with a major spike as the array increases in size from 2MB and greater. This spike indicates that the array no longer fits in L3 and is now reading from the main memory. Each plateau corresponds to a cache hierarchy, with the spike indicating that the array is now being read from the main memory. Therefore, it can be safely assumed that the LLC is between 2MB – 4MB; which is confirmed by the specifications to be 3MB.



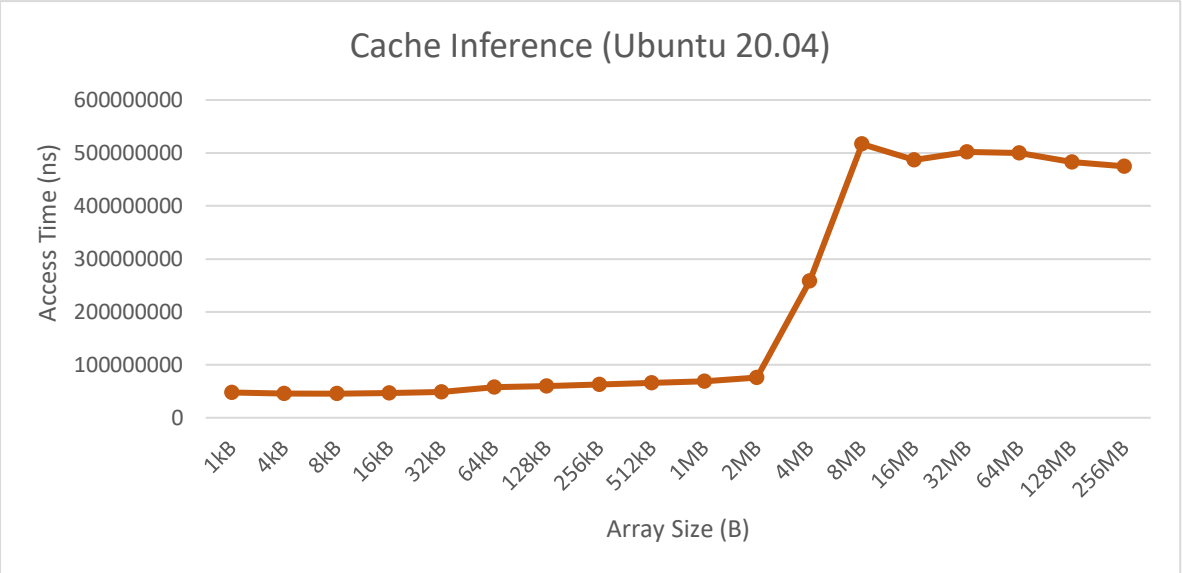
2.1.2 Virtual Machines

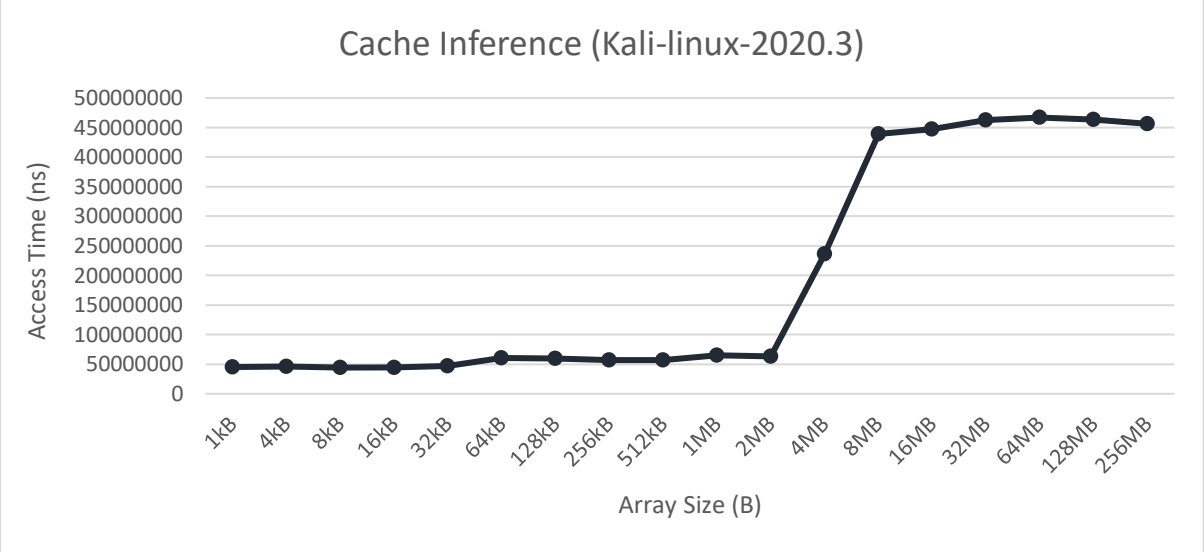
The following tests were conducted on an HP Envy x360 15 running both an Ubuntu 20.04 and Kali Linux 2020.3 virtual machine using an AMD Ryzen 7 processor.

HP Envy x360 15, Ubuntu 20.04 Specifications	
Architecture	x86_64
Address Sizes	48 bits physical, 48 bits virtual
CPU(s)	4
Model name	AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx
L1d cache	128 KiB
L1i cache	256 KiB
L2 cache	2 MiB
L3 cache	4 MiB

HP Envy x360 15, Kali Linux 2020.3 Specifications	
Architecture	x86_64
Address Sizes	48 bits physical, 48 bits virtual
CPU(s)	2
Model name	AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx
L1d cache	64 KiB
L1i cache	128 KiB
L2 cache	1 MiB
L3 cache	4 MiB

Both virtual machines produced similar graphs, where, from the results, it is reasonable to assume that the LLC is between 2MB and 8MB; however, when comparing with the Kali Linux specifications it appears that the overestimated the size of the LLC. This could mean that a different approach in measuring the LLC may be needed for AMD processors.

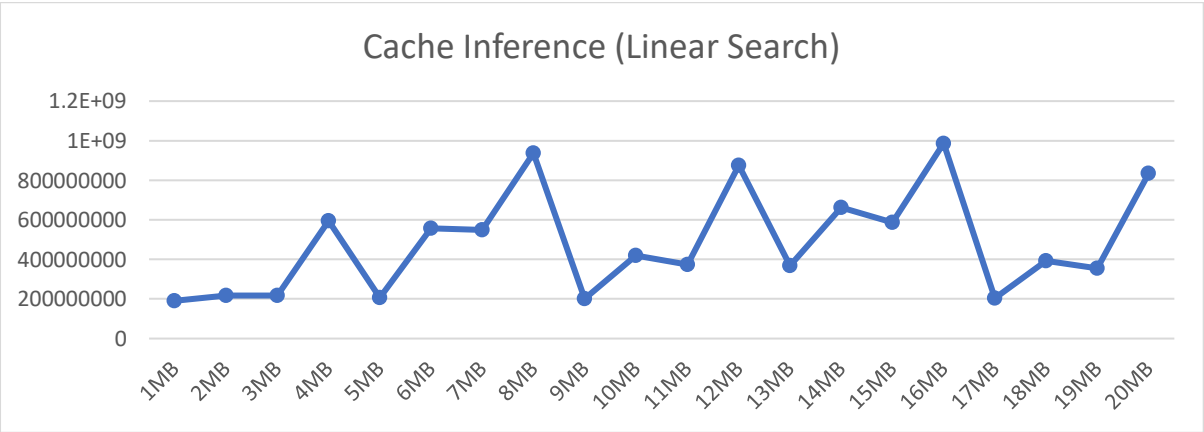




2.2 Alternative Parameters and Configurations[c]

2.2.1 Linear Array Size Increase

From the results gathered from an exponential search, we can further refine where exactly the LLC is by linearly increasing the size of the array. The program now measures the access times for sizes of 1MB to 20MB. From these results, it can be inferred that the LLC is around 3MB; however, it is interesting to observe the resulting sawtooth pattern that occurs as it begins reading from the main memory. Possible causes could be due to garbage collection[d] or thrashing.



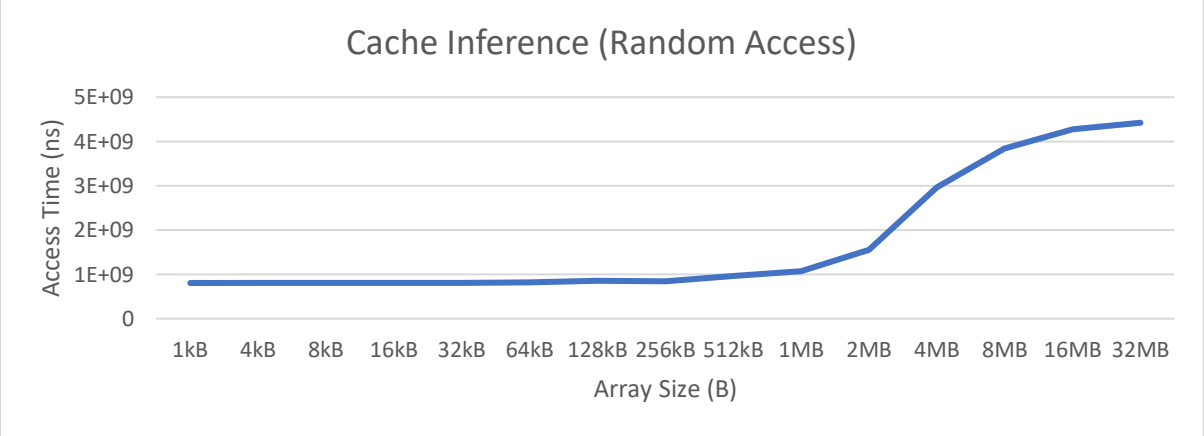
[c] All changes made to the program were tested only using the HP ZBook 17 with WSL

[d] The JVM continuously allocates memory on the heap as new objects are created in execution time. These short-lived objects are then made available for garbage collection. The continuous drop seen in the graph is the garbage collector finishing.[8]

2.2.2 Random Array Access

Another consideration was the possible effect that prefetching may have had on the performance and, therefore, the results. The CPU would predict what datum will be accessed next, causing the CPU to speculatively execute following operations in advance before it is needed. Consequently, this cause allows the program to minimize the wait times potentially affecting the access timings due to the program implementing sequential array access.[e] To account for the potential effect prefetching has, the program was altered so the read/write loop performed a random-access using Java's Random class, as opposed to a linear search.

From the results, we can observe the distinct increase in access time after 2MB, once more highly indicating that the program has passed the LLC and is now reading from main memory, and we can infer that the LLC is around this size. Perhaps configuring the program to run both linear array size and random array access at the same time would give a closer estimate of the correct cache sizes.



Conclusion

Understanding how to infer the size of the LLC gives an insight into how side channels can take advantage of knowing when the victim is accessing different cache lines through measuring the access time of data. Perhaps writing the program in a low-level language like C would present closer, accurate results (unlike Java and the potential side-effects from the garbage collector). Overall, this program was able to provide an insightful look into the relationship between array size and access time in identifying the structure of a modern CPU.

[e] Refer to Techopedia, *prefetching*. https://www.techopedia.com/definition/32421/prefetching

References

[1] Phillips, G. (2021, February 17). *How Does CPU Cache Work? What Are L1, L2, and L3 Cache?* Retrieved from MakeUseOf: https://www.makeuseof.com/tag/what-is-cpu-cache/

[2] Lutkevich, B. (n.d.). *cache memory*. Retrieved from TechTarget: https://searchstorage.techtarget.com/definition/cache-memory

[3] Mushtaq, M., Mukhtar M.A., Lapotre, V., Bhatti, M., Gogniat, G. *Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA*. Information Systems, Elsevier, In press, ⟨10.1016/j.is.2020.101524⟩. ⟨hal-02537540⟩

[4] Ostrovsky, I. (2010, January 19). *Gallery of Processor Cache Effects*. Retrieved from Igor Ostrosky Blogging: https://igoro.com/archive/gallery-of-processor-cache-effects/

[5] EventHelix. (2017, July 2017). *Why software developers should care about CPU caches*. Retrieved from Medium: https://medium.com/software-design/why-software-developers-should-care-about-cpu-caches-8da04355bb8a

[6] Fiege, M. (2019, February 7). *Arrays, Linked Lists, and Big O Notation*. Retrieved from Medium: https://medium.com/@mckenziefiege/arrays-linked-lists-and-big-o-notation-486727b6259b

[7] Heinrichs, M. (2021, January 30). *Java and the modern CPU, Part 1: Memory and the cache hierarchy*. Retrieved from Java Magazine: https://blogs.oracle.com/javamagazine/post/java-and-the-modern-cpu-part-1-memory-and-the-cache-hierarchy

[8] Beiske, K. (2015, January 22). *Understanding the Memory Pressure Indicator*. Retrieved from elastic: https://www.elastic.co/blog/found-understanding-memory-pressure-indicator