Side Channel Attacks

Jakob Millen
1507831

Introduction

This report shows the analysis and testing of the CPU cache to find the size of the Last Level Cache (LLC) using a timing side-channel attack on two different Linux systems, a Raspberry Pi 4 and a Virtual Machine (VM) running Ubuntu 22.04. By measuring the time taken to access different sizes of memory arrays, we can understand the cache sizes and the performance characteristics of these systems.

Methodology

Approach:

My approach involves accessing the memory arrays of varying sizes and measuring the access time. The sizes range from 1KB to 256MB, increasing exponentially. The objective of this approach is to identify the size of where a significant increase in access time has occurred, indicating that the array size exceeds the current cache capacity, leading to frequent cache misses and eventually accessing the slower main memory.

Steps:

- **Array Allocation:** Memory arrays of sizes from 1KB to 256KB are allocated
- **Access Pattern:** Each element of the array is accessed using a stride of 64 bytes, which matches the CPU's cache line size.
- **Warm-up Phase:** The cache is warmed up by accessing the array multiple times before the actual timing is measured to ensure the cache is populated and is not interfered with by any background process.
- **Timing Measurements:** The time taken to access each array size is measured in nanoseconds over multiple iterations and averaged to smooth out anomalies.

Environment:

- Raspberry Pi 4:

The Raspberry Pi 4 runs on an ARM Cortex-A72 CPU. The Cortex CPU uses an L1 cache of 32 KB and an L2 cache of 512 KB. Although the Raspberry Pi doesn't have an L3 cache the side-channel attack algorithm can still be used to verify cache size with L1 and L2. The operating system used by the Raspberry Pi is Debian GNU/Linux.

- Virtual Machine (Ubuntu 22.04)

The Virtual Machine (VM) is running inside a Windows 10 system that uses an Intel I5 9600k CPU. The I5 CPU uses an L1 Cache of 32 KB a L2 cache of 256 KB and a L3 cache of 9MB.

Code Design:

The program was created using Java and reads different array sizes and times the memory accesses to infer the cache sizes, within the code is the algorithm which was inspired by Igor Ostrosky's version. The algorithm loops over the array, incrementing every 64th byte which aligns with the cache line that the CPU uses.

```
int steps = 64 * 1024 * 1024; // Arbitrary number of steps
int lengthMod = arr.Length - 1;
for (int i = 0; i < steps; i++)
{
    arr[(i * 16) & lengthMod]++; // (x & lengthMod) is equal to (x % arr.Length)
}
```

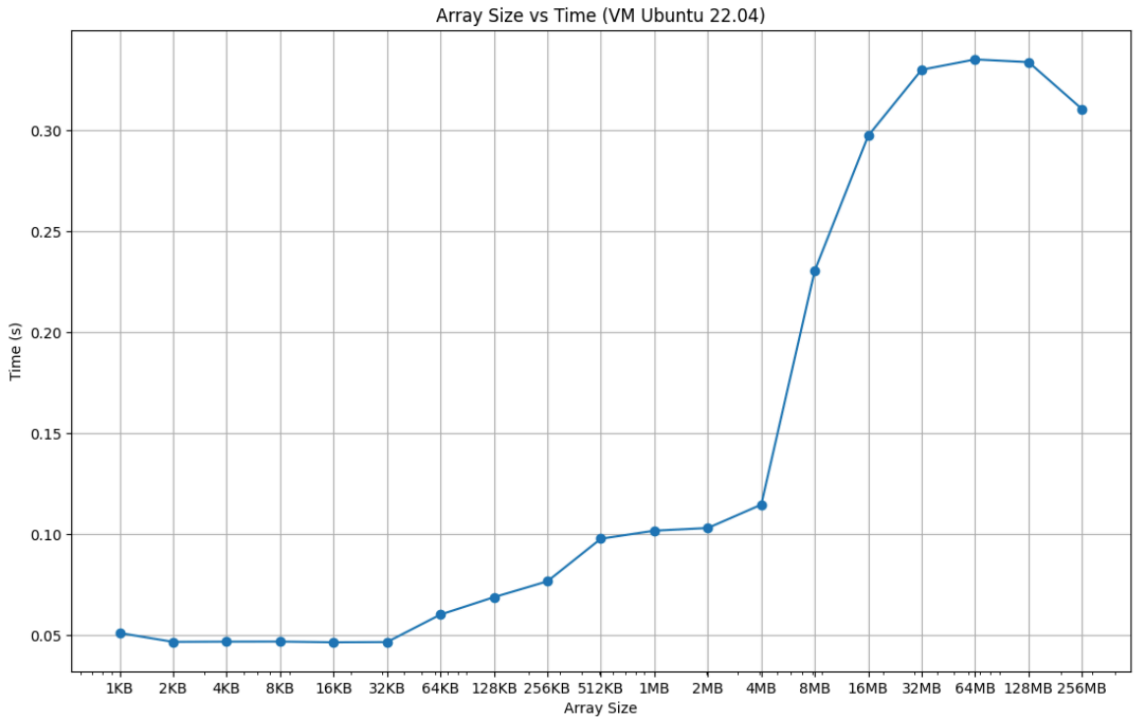
This algorithm is the one that Igor uses and which I was inspired by. In Igor's version, he uses an Integer array with the 16 representing his 64 bytes. In my version, I opted for a Byte array and used 64 bytes although it wouldn't make a significant difference a byte array over an integer array can provide more detailed testing and reduced overhead. This can potentially result in testing larger arrays without consuming excessive memory, making it easier to push the limits of the cache.

In my design, I also implemented a warm-up phase before testing this will hopefully ensure that the system is in a steady state before starting the actual measurements.

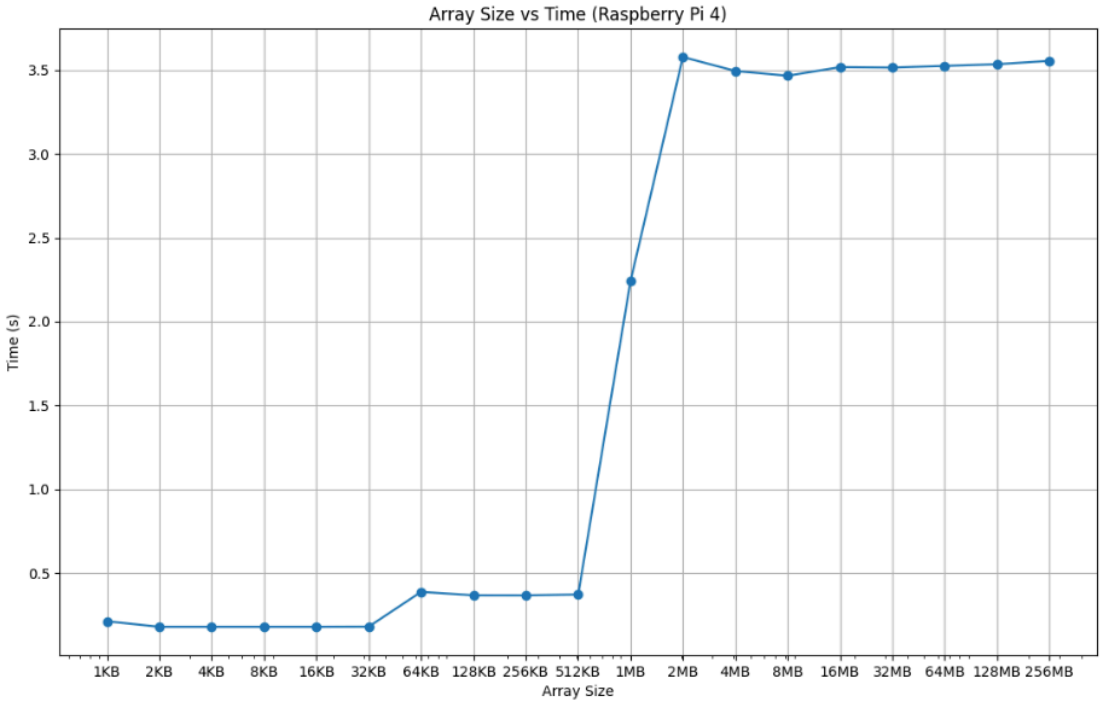
When testing the program its initial access to the CPU cache may be filled with other data which although minimal it can obscure the result. For a consistent measurement warming up the cache by running the memory access pattern a few times before the test can help ensure that the data is consistently in the cache. This makes the measurements more reflective of the actual cache performance rather than the initial loading overhead.

Results:

VM (Ubuntu 22.04)



Raspberry Pi 4



Analysis:

Observation of Raspberry Pi 4:

The initial phase from 1KB to 32KB, is very consistent and shows low access time indicating efficient handling by the L1 cache. Moving from 64KB to 512KB shows very consistent access times similar to the L1 cache. These low access times confirm the cache size specified by the Raspberry Pi and show a significant spike after the L2 cache has filled verifying that there is no L3 cache and the main memory is now being used.

Observation of VM (Ubuntu 22.04):

In the initial phase from 1KB to 32KB, we can see a consistently low access time, resembling the L1 cache. Past 32 KB there is a steady increase in access time as the L2 cache starts to fill up and once reaching 256 KB there is a noticeable spike in access time indicating that the L3 cache is now being used. From 512 KB to 4MB we can see a similar pattern to the handling of the L1 and L2 caches. A noticeable spike in time is seen at 4MB, which is earlier than the expected 9MB L3 cache size. This suggests that the actual effective cache capacity is lower due to factors like associativity and cache management overhead. This isn't a surprise as background processes are running which would interfere with the results. Beyond 8MB we can clearly see that the main memory is heavily being relied on for these larger sizes.

Conclusion:

The timing side-channel approach effectively reveals the cache behavior and capacity across different systems. The testing confirms the significant role of the cache levels in memory access performance for both the Raspberry Pi and the Virtual Machine. Noticeable spikes in the access time align with the limits of the cache lines of each system. Although the results from the VM were not as clear as the Raspberry Pi, it still acts as an indication as to how accurate the algorithm is.

To further improve the accuracy of the VM some issues need to be addressed. Factors like the virtual machines having to share resources can lead to unpredictable performance variations and can impact the timing measurements. Cache eviction strategies that modern CPUs have can also affect how data is cached and evicted, affecting the access times as the working set size approaches or exceeds cache capacity. Most important I think is the system load, with background processes and system-level activities on the host machine this can interfere with the timing measurements, leading to variations in the results. System load impacts are more noticeable in VM environments due to sharing resources with the host.