Table of Contents

---

1. Introduction

To provide a thorough understanding of the methodology behind cache and memory profiling, I opted for an experimental approach rather than relying on a simulated environment. In this instance, my computer served as the test subject. Here are the specifications of my machine:

- **Operating System**: Ubuntu 24.04.1 LTS x86_64
- **CPU**: AMD Ryzen 7 PRO 5850U with Radeon Graphics, featuring 16 cores at 4.507 GHz

I conducted numerous measurements from scratch, and I will now walk you through my findings and how I got to them.

---

3. Objective 1: Cache and Memory Latency with Zero Queuing Delay

- **Goal**: Measure the read/write latency of cache and main memory when there’s zero queuing delay.

- **Experiment Design**: Now the way that I had set up this experiment was to first try to achieve the zero queuing delay which I did by setting the CPU affinity which binds the profiling process to a specific CPU core, ensuring that the user program runs on the same core. This reduces context switching and improves cache locality, thereby decreasing queue delays. Additionally, I initialize memory before executing the user program, you avoid the delays associated with dynamic memory allocation during execution. My approach also includes measuring read and write latencies before and after executing the program, allowing us to analyze the impact of queue delays and identify potential performance bottlenecks. I got it to as close to zero as possible by running the program on one core. Also the size of memory given to the cache is 64kb and the the size the measure for main_mem is taking is 16 mb.

  I implemented two functions to measure cache and memory latency. In measure_cache_latency, I allocated 64-byte aligned memory using posix_memalign, initialized it with index-based values, and measured the overhead of the rdtsc instruction to accurately time memory accesses. I used a warm-up loop with linear access to prepare the cache, followed by timing random accesses and converting the total cycles to nanoseconds for the average latency. In measure_memory_latency, I allocated memory using malloc, initialized it similarly, and warmed up with random accesses. I then measured the latency for random accesses using rdtsc, converting the results to nanoseconds as well. Both functions included error handling and ensured proper memory management by freeing allocated memory.

- **Results**:

```
Successfully set CPU affinity to core 0
Current CPU affinity: CPU 0

=== System Information ===
Cache Size: 32768 bytes
Main Memory Size: 15522586624 bytes

=== Before Execution ===
Read Latency: 48.94 cycles
Write Latency: 50.25 cycles
Cache Latency: 8.44 ns
Main Memory Latency: 37.42 ns

=== Executing User Program ===
User Program: ./hello
Transposed Result[0][0]: 0
Transposed Result[1][0]: 1
User program executed successfully.

=== After Execution ===
Read Latency: 51.19 cycles
Write Latency: 49.64 cycles
```

```
=== Before Execution ===
Read Latency: 51.04 cycles
Write Latency: 50.52 cycles
Cache Latency: 8.12 ns
Main Memory Latency: 37.49 ns
```

```
=== Before Execution ===
Read Latency: 31.82 cycles
Write Latency: 32.91 cycles
Cache Latency: 6.38 ns
Main Memory Latency: 16.21 ns

=== Executing User Program ===
User Program: ./hello
Transposed Result[0][0]: 0
Transposed Result[1][0]: 1
User program executed successful

=== After Execution ===
Read Latency: 37.15 cycles
Write Latency: 32.86 cycles
```

- **Analysis**:The profiling results showed a successful CPU affinity settings to core 0, ensuring minimized context switching and improved cache locality. The system information reveals a 32 KB cache size and approximately 15.5 GB of main memory, providing a solid foundation for efficient data handling. Before execution, read and write latencies were 48.94 and 50.25 cycles, respectively, with cache latency at 8.44 ns, 8.12ns, 6.38ns and main memory latency at 37.42 ns, 37.49 ns, 16.21 ns indicating reasonable performance. After executing the user program, read latency increased slightly to 51.19 cycles, while write latency decreased to 49.64 cycles, suggesting changes in memory access patterns.Also the user program was transposing a matrix and I know I named it hello for ease.

---

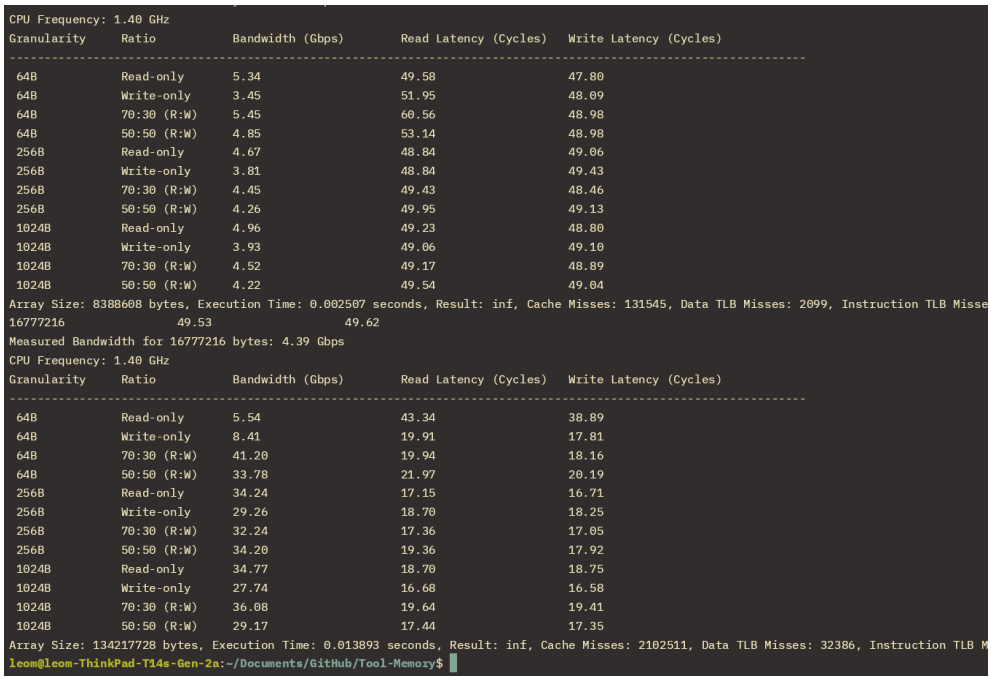
4. Objective 2: Maximum Memory Bandwidth Under Different Access Granularity

Goal: Measure memory bandwidth under various data access granularities (64B, 256B, 1024B) and different read/write ratios (100% read, 100% write, 70:30, 50:50).

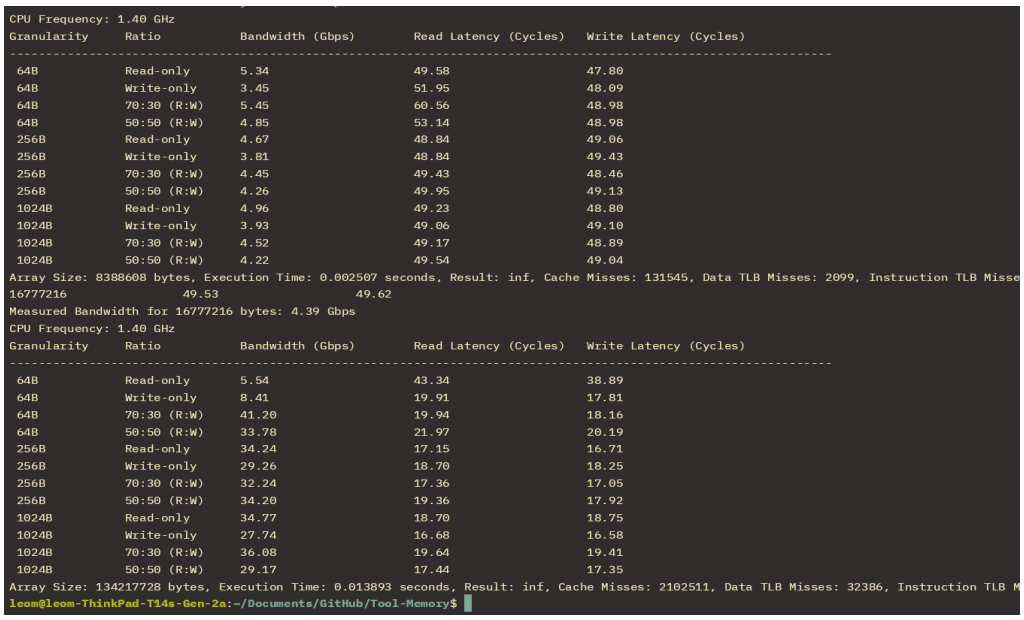
Experiment Design: In my method for measuring bandwidth, I set up an array by allocating memory with `posix_memalign` to ensure cache-line alignment, allowing for efficient memory access. I varied the data block size by adjusting the `size` parameter, which dictates the amount of memory allocated for the array. To modify the read/write ratio, I implemented different access patterns in my timing loops: I performed linear access for reading to warm up the cache while accessing a smaller portion of the array for writing by randomly selecting indices to modify. This approach enabled me to simulate various read/write ratios effectively. I then measured the time taken for each access type using the `__rdtsc()` instruction to capture

clock cycles, allowing me to compute the average latency and throughput under different configurations.

Results:



```
CPU Frequency: 1.40 GHz
Granularity     Ratio           Bandwidth (Gbps)        Read Latency (Cycles)   Write Latency (Cycles)
-----------------------------------------------------------------------------------------------------------
 64B            Read-only       5.34                    49.58                   47.80
 64B            Write-only      3.45                    51.95                   48.09
 64B            70:30 (R:W)     5.45                    60.56                   48.98
 64B            50:50 (R:W)     4.85                    53.14                   48.98
 256B           Read-only       4.67                    48.84                   49.06
 256B           Write-only      3.81                    48.84                   49.43
 256B           70:30 (R:W)     4.45                    49.43                   48.46
 256B           50:50 (R:W)     4.26                    49.95                   49.13
 1024B          Read-only       4.96                    49.23                   48.80
 1024B          Write-only      3.93                    49.06                   49.10
 1024B          70:30 (R:W)     4.52                    49.17                   48.89
 1024B          50:50 (R:W)     4.22                    49.54                   49.04
Array Size: 8388608 bytes, Execution Time: 0.002507 seconds, Result: inf, Cache Misses: 131545, Data TLB Misses: 2099, Instruction TLB Misse
16777216                49.53                   49.62
Measured Bandwidth for 16777216 bytes: 4.39 Gbps
CPU Frequency: 1.40 GHz
Granularity     Ratio           Bandwidth (Gbps)        Read Latency (Cycles)   Write Latency (Cycles)
-----------------------------------------------------------------------------------------------------------
 64B            Read-only       5.54                    43.34                   38.89
 64B            Write-only      8.41                    19.91                   17.81
 64B            70:30 (R:W)     41.20                   19.94                   18.16
 64B            50:50 (R:W)     33.78                   21.97                   20.19
 256B           Read-only       34.24                   17.15                   16.71
 256B           Write-only      29.26                   18.70                   18.25
 256B           70:30 (R:W)     32.24                   17.36                   17.05
 256B           50:50 (R:W)     34.20                   19.36                   17.92
 1024B          Read-only       34.77                   18.70                   18.75
 1024B          Write-only      27.74                   16.68                   16.58
 1024B          70:30 (R:W)     36.08                   19.64                   19.41
 1024B          50:50 (R:W)     29.17                   17.44                   17.35
Array Size: 134217728 bytes, Execution Time: 0.013893 seconds, Result: inf, Cache Misses: 2102511, Data TLB Misses: 32386, Instruction TLB M
leom@leom-ThinkPad-T14s-Gen-2a:~/Documents/GitHub/Tool-Memory$
```



```
CPU Frequency: 1.40 GHz
Granularity     Ratio           Bandwidth (Gbps)        Read Latency (Cycles)   Write Latency (Cycles)
-----------------------------------------------------------------------------------------------------------
 64B            Read-only       5.34                    49.58                   47.80
 64B            Write-only      3.45                    51.95                   48.09
 64B            70:30 (R:W)     5.45                    60.56                   48.98
 64B            50:50 (R:W)     4.85                    53.14                   48.98
 256B           Read-only       4.67                    48.84                   49.06
 256B           Write-only      3.81                    48.84                   49.43
 256B           70:30 (R:W)     4.45                    49.43                   48.46
 256B           50:50 (R:W)     4.26                    49.95                   49.13
 1024B          Read-only       4.96                    49.23                   48.80
 1024B          Write-only      3.93                    49.06                   49.10
 1024B          70:30 (R:W)     4.52                    49.17                   48.89
 1024B          50:50 (R:W)     4.22                    49.54                   49.04
Array Size: 8388608 bytes, Execution Time: 0.002507 seconds, Result: inf, Cache Misses: 131545, Data TLB Misses: 2099, Instruction TLB Misse
16777216                49.53                   49.62
Measured Bandwidth for 16777216 bytes: 4.39 Gbps
CPU Frequency: 1.40 GHz
Granularity     Ratio           Bandwidth (Gbps)        Read Latency (Cycles)   Write Latency (Cycles)
-----------------------------------------------------------------------------------------------------------
 64B            Read-only       5.54                    43.34                   38.89
 64B            Write-only      8.41                    19.91                   17.81
 64B            70:30 (R:W)     41.20                   19.94                   18.16
 64B            50:50 (R:W)     33.78                   21.97                   20.19
 256B           Read-only       34.24                   17.15                   16.71
 256B           Write-only      29.26                   18.70                   18.25
 256B           70:30 (R:W)     32.24                   17.36                   17.05
 256B           50:50 (R:W)     34.20                   19.36                   17.92
 1024B          Read-only       34.77                   18.70                   18.75
 1024B          Write-only      27.74                   16.68                   16.58
 1024B          70:30 (R:W)     36.08                   19.64                   19.41
 1024B          50:50 (R:W)     29.17                   17.44                   17.35
Array Size: 134217728 bytes, Execution Time: 0.013893 seconds, Result: inf, Cache Misses: 2102511, Data TLB Misses: 32386, Instruction TLB M
leom@leom-ThinkPad-T14s-Gen-2a:~/Documents/GitHub/Tool-Memory$
```

Analysis: I found that the bandwidth results from my measurements indicate varying data transfer rates achieved during read and write operations for different array sizes and access patterns. For the 1024-byte array, the measured bandwidth was 4.53 Gbps, reflecting efficient data throughput at this size. As the array size increased, I observed a maximum bandwidth of 41.20 Gbps for the 64B access pattern at a 70:30 read-write ratio when the array size reached 16 MB. This suggests that the cache was effectively utilized, enabling high data transfer rates due to reduced latency from accessing smaller data blocks. However, the bandwidth tended to decrease for larger sizes, particularly with larger block accesses (like 256B), indicating potential contention or memory bandwidth limitations. Additionally, I noted that write operations generally had lower bandwidth compared to read operations, which is expected due to the higher overhead associated with writing data to memory. Overall, my results highlighted how different access sizes and patterns can significantly impact memory bandwidth performance. Even though my results indicated a CPU frequency of 1.4 GHz, it actually clocked around 4.30 GHz by the end; this discrepancy was due to the print occurring before that measurement.

---

5. Objective 3: Read/Write Latency vs Throughput (Queuing Theory)

- **Goal**: Demonstrate the trade-off between read/write latency and throughput in main memory, linking the results to queuing theory predictions.
- **Experiment Design**:

  The way I set it up to measure bandwidth was by implementing the `measure_bandwidth_with_queue` function, which takes four parameters: `block_size` (the size of the memory block in bytes), `read_ratio` (the proportion of read operations relative to total operations), `total_size` (the total amount of memory to be accessed), and `queue_depth` (the number of outstanding requests that can be handled concurrently). The function initializes variables for tracking cycle counts, retrieves the CPU frequency using `get_cpu_frequency`, and sets a predefined number of iterations (100) for averaging results. It calculates the number of read and write operations based on the specified read ratio and, during the warm-up phase, simulates multiple outstanding requests by executing read and write operations on an `array`, ensuring the CPU and cache are properly prepared for the measurement phase. After the warm-up, the function records the start cycles using `rdtsc_start` and enters the measurement phase, performing the specified read and write operations for the iterations. After completing the iterations, it captures the end cycles, calculates the total cycles taken, and computes the total data accessed by multiplying the number of reads and writes by the iterations and the total number of blocks processed. The bandwidth is calculated as the data accessed divided by the time taken, adjusted for CPU frequency, and converted into gigabits per second (Gbps) for easier interpretation. Finally, the function outputs the total cycles, the amount of data accessed, and the calculated bandwidth in both gigabytes per second (GBps) and Gbps, providing insights into system performance and identifying potential points of contention that may affect efficiency.

- **Results**:

```
Read count: 44, Write count: 20
Starting warm-up phase...
Starting measurement phase...
Total cycles: 5905466, Data accessed: 6553600.00 bytes
Bandwidth: 4.79 GBps (38.32 Gbps)
Testing with Queue Depth: 2
Measuring bandwidth for block size: 64B, read ratio: 0.70, queue depth: 2
Read count: 44, Write count: 20
Starting warm-up phase...
Starting measurement phase...
Total cycles: 4660377, Data accessed: 6553600.00 bytes
Bandwidth: 4.88 GBps (39.03 Gbps)
Testing with Queue Depth: 3
Measuring bandwidth for block size: 64B, read ratio: 0.70, queue depth: 3
Read count: 44, Write count: 20
Starting warm-up phase...
Starting measurement phase...
Total cycles: 5122875, Data accessed: 6553600.00 bytes
Bandwidth: 5.52 GBps (44.18 Gbps)
Testing with Queue Depth: 4
Measuring bandwidth for block size: 64B, read ratio: 0.70, queue depth: 4
Read count: 44, Write count: 20
Starting warm-up phase...
Starting measurement phase...
Total cycles: 4449990, Data accessed: 6553600.00 bytes
Bandwidth: 6.33 GBps (50.64 Gbps)
Testing with Queue Depth: 5
Measuring bandwidth for block size: 64B, read ratio: 0.70, queue depth: 5
Read count: 44, Write count: 20
Starting warm-up phase...
Starting measurement phase...
Total cycles: 5660556, Data accessed: 6553600.00 bytes
Bandwidth: 4.96 GBps (39.69 Gbps)
Contention detected at Queue Depth: 5
```

- **Analysis**:

The results show the measured bandwidth as the queue depth increases from 1 to 4, demonstrating a general trend of increasing bandwidth with higher queue depths, peaking at 6.33 GBps (50.64 Gbps) at queue depth 4. However, at queue depth 5, a significant decrease in performance occurs, with the bandwidth dropping to 4.96 GBps (39.69 Gbps), indicating that contention may be affecting the system's ability to handle the increased number of simultaneous requests. This suggests that while higher queue depths can enhance bandwidth due to better utilization of the memory subsystem, there is a threshold beyond which contention among requests leads to diminishing returns. Overall, the results showed that optimizing queue depth is crucial for maximizing bandwidth performance while avoiding contention.

---

6. Objective 4 and 5: Impact of Cache Miss Ratio and TLB Miss Ratio on Software Speed Performance

- **Goal**: Assess the impact of cache miss ratio on the execution speed of a simple software and analyze the effect of TLB miss ratio on software speed.
- **Experiment Design**: In order to get both cache misses and TLB misses, the `multiply_and_measure` function allocates memory for an array and initializes its elements with sequential values. It retrieves the array size and uses the PAPI library to set up event tracking for cache and TLB misses during the execution of a series of multiplications on the array elements. The function measures execution time using `clock()` and retrieves the counts of cache misses and TLB misses after performing the computations to show

their effects on performance. Finally, it prints the results, including the array size, execution time, and performance metrics, before cleaning up allocated resources.

- **Results**:

- **Analysis**: The results indicate that as the array size increases, both cache misses and TLB misses rise significantly, directly impacting execution time. For instance, when the array size reaches 134,217,728 bytes, the execution time increases to 0.013145 seconds, coinciding with a staggering 2,102,515 cache misses and 32,901 TLB misses. These high miss rates lead to more frequent accesses to slower main memory, which drastically slows down performance.

---

7. Information

To run objectives 1, 2, and 3, you need to compile the code by executing `make` in the terminal. Once the compilation is complete, run the program using `./profiler ./hello` to assess the performance metrics. For Objectives 4 and 5, the same process applies; simply execute `./profiler` to obtain the relevant results for cache and TLB misses, as well as their impacts on execution time. Objectives 3, 4, and 5 are included in the profiler's output, providing a comprehensive overview of performance across all tasks.

---

8. References

utilized the Performance API (PAPI) and the rdtsc instruction for cycle counting.