Course Project #1: Cache and Memory Performance Profiling
Due 25 September 2024

The objective of this project is to gain deeper understanding of cache and memory hierarchy in modern computers. You should design a set of experiments that will quantitatively reveal the following:

1. the read/write latency of cache and main memory when the queue length is zero (i.e., zero queuing delay)

   Experiment 1 is designed to measure read and write latencies of cache and main memory when there is zero queuing delay, meaning no other memory requests are pending. The Python code simulates this scenario by issuing memory reads and writes sequentially, without introducing any additional load or contention on the memory subsystem. Each memory access is handled immediately, ensuring that there is no waiting period for other memory accesses—this fulfills the requirement of having a zero queue length.

   The experiment is implemented in Python, where memory access patterns are simulated to measure both read and write latencies across different buffer sizes. The code first sets up a memory buffer, simulating data sizes that correspond to cache and main memory levels. For each buffer size, memory-mapped I/O is used to perform repeated read and write operations in a controlled loop. The code ensures that each operation is sequential and isolated, reflecting a scenario with zero queuing delay, meaning no other memory operations interfere with the current one. By measuring the time taken for these operations, the experiment captures the latency purely attributed to the buffer size and the underlying memory hierarchy. The data collected is then used to plot the relationship between buffer size and latency, giving insight into how memory access times vary as the system moves from accessing cache to main memory.

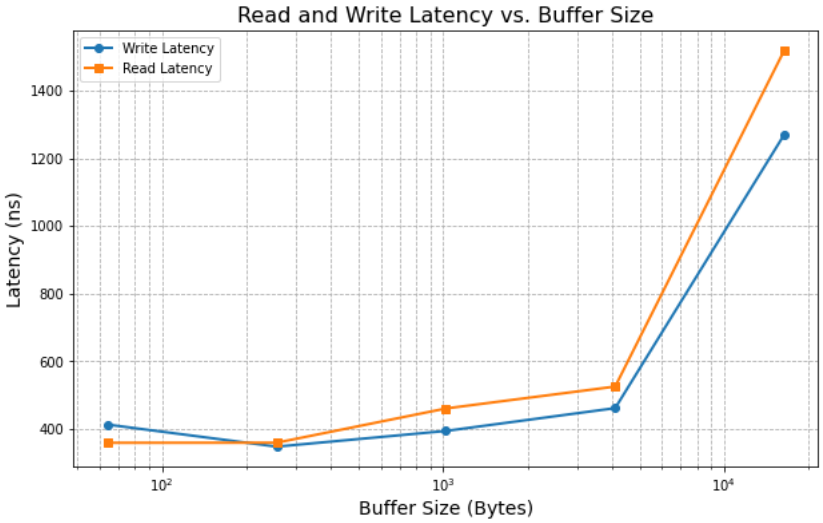


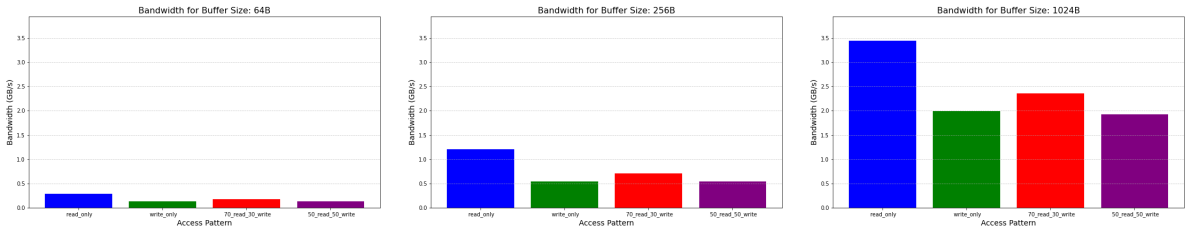
Figure 1: Read and Write Latency vs. Buffer Size

From the results, we observe that for smaller buffer sizes (64B, 256B), which correspond to cache levels, both read and write latencies remain low around 350–400 ns. This reflects the fast access times of the cache. As the buffer size increases, especially beyond 4096B, we see a significant jump in latency, with the largest buffer (16384B) reaching over 1200 ns for writes and 1500 ns for reads. This reflects the transition from accessing cache to main memory, which is inherently slower. The experiment shows how the memory hierarchy influences access times, specifically isolating the intrinsic latencies of cache and main memory. By eliminating queuing delays, it focuses purely on the hardware's raw performance, providing a clear understanding of how each memory level contributes to overall access time without interference from external system load.

2. the maximum bandwidth of the main memory under different data access granularity (i.e., 64B, 256B, 1024B) and different read vs. write intensity ratio (i.e., read-only, write-only, 70:30 ratio, 50:50 ratio)

   The goal of Experiment 2 was to measure the maximum bandwidth of the main memory under different data access granularities (64B, 256B, 1024B) and read vs. write intensity ratios (read-only, write-only, 70:30 ratio, and 50:50 ratio). Memory bandwidth refers to the amount of data transferred per second, and this experiment aimed to explore how different data sizes and patterns of access impact the overall performance of the system's memory. The experiment focused on simulating these memory access patterns and calculating the bandwidth for each combination of buffer size and read/write ratio.

   The Python code created memory buffers of different sizes (64B, 256B, and 1024B) to simulate memory accesses at different granularities. These buffer sizes represent typical access patterns in computer systems, with smaller sizes fitting into cache lines and larger sizes potentially spilling into main memory. The code simulated different access patterns, including read-only, write-only, and mixed read/write ratios (70% reads and 30% writes, as well as a 50:50 split). For each buffer size and access pattern, the code measured how much data was transferred and calculated the bandwidth in gigabytes per second (GB/s).



(a) 64B (b) 256B (c) 1024B

Figure 2: Bandwidth Plots of 64B, 256B, and 1024B

From the terminal output, we observe that the bandwidth increases as the buffer size grows. For example, with a buffer size of 64B, the maximum bandwidth achieved for read-only operations is 0.29 GB/s, whereas for a buffer size of 1024B, the bandwidth increases significantly to 3.44 GB/s. This indicates that larger buffer sizes allow more efficient use of memory bandwidth, as more data can be processed in a single operation. This trend is consistent across all access patterns.

For the access patterns, read-only operations consistently achieved higher bandwidth compared to write-only operations, regardless of buffer size. For example, with a 1024B buffer, the bandwidth for read-only access is 3.44 GB/s, while for write-only access, it is significantly lower at 1.99 GB/s. In mixed patterns like the 70:30 read/write ratio, the bandwidth tends to lie between the read-only and write-only extremes, reflecting the combination of reads and writes in the operation. For instance, with a 1024B buffer, the bandwidth for the 70:30 ratio is 2.36 GB/s, falling between the values for read-only and write-only.

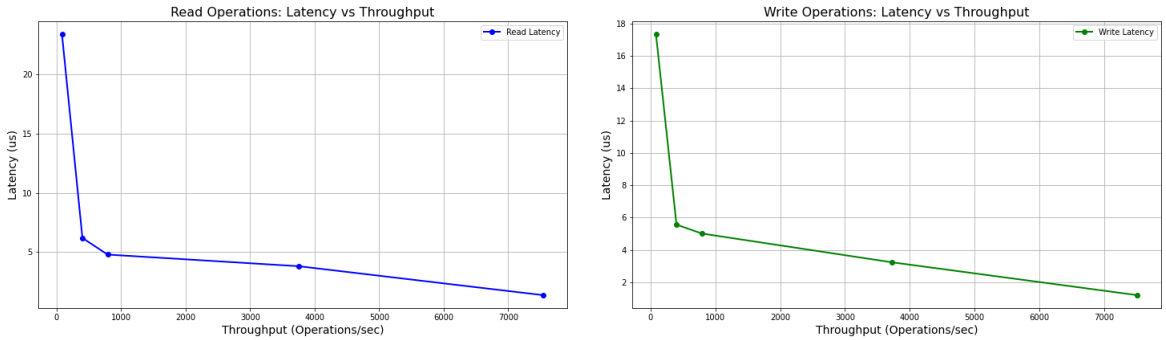
Smaller buffer sizes, such as 64B, show relatively low bandwidth values across all access patterns. The bandwidth for write-only access with a 64B buffer is 0.14 GB/s, and even for read-only access, it only reaches 0.29 GB/s. This is because smaller buffer sizes require more frequent memory accesses to transfer the same amount of data, reducing overall efficiency and resulting in lower bandwidth.

In conclusion, the results show that larger buffer sizes and read-heavy access patterns provide the highest memory bandwidth, while smaller buffers and write-heavy patterns result in lower bandwidth. This shows that the larger and more contiguous data transfers are more efficient, and memory systems seem to be optimized for reads over writes.

3. the trade-off between read/write latency and throughput of the main memory to demonstrate what the queuing theory predicts

   The purpose of Experiment 3 was to simulate and quantify the trade-off between read/write latency and throughput in main memory. The goal was to demonstrate queuing theory predictions, which suggest that as the system's throughput increases, latency also increases due to queuing delays and system saturation. This experiment specifically aimed to measure how latency changes as the throughput varies for both read and write operations.

   The Python code simulates read and write memory operations at different throughput levels. The throughput levels tested were 100, 500, 1000, 5000, and 10000 operations per second.



(a) Read (b) Write

Figure 3: Latency vs. Throughput Graphs

The results of the experiment demonstrate the expected relationship between throughput and latency. For read operations, as the throughput increased from 82.64 ops/sec to 7542.95 ops/sec, the latency decreased from 23.37 microseconds to 1.38 microseconds. Similarly, for write operations, as the throughput increased from 82.40 ops/sec to 7512.20 ops/sec, the latency dropped from 17.34 microseconds to 1.21 microseconds. This indicates that as the system processes more operations per second, the average time taken to complete each operation decreases. Both read and write operations follow a similar trend, with higher throughput levels leading to significantly lower latencies.

These results align with queuing theory predictions, which suggest that a system handling a higher volume of operations becomes more efficient, reducing latency as it processes more requests per second. At lower throughput levels like 100 ops/sec, the system is underutilized, leading to relatively higher latency due to the inefficiency of handling fewer operations. As throughput increases such as 5000–10000 ops/sec, the system operates closer to full capacity, optimizing the processing time per operation and lowering the latency. This trade-off between latency and throughput is a key consideration in system performance tuning, as maximizing throughput helps reduce delays, provided the system does not become saturated.

4. the impact of cache miss ratio on the software speed performance (the software is supposed to execute relatively light computations such as multiplication)

   The goal of Experiment 4 was to examine how the cache miss ratio affects software speed performance during light computations, specifically matrix multiplication. As cache misses increase, access to slower main memory becomes more frequent, resulting in slower execution times. This experiment simulates the impact of cache misses by performing matrix multiplications with matrices of increasing sizes, where larger matrices are more likely to exceed cache capacity, leading to higher cache miss ratios.

   The Python code simulates the experiment by performing matrix multiplications on matrices of varying sizes, ranging from 100x100 to 3000x3000. Smaller matrices like 100x100 are expected to fit into the CPU cache, resulting in fast execution times with minimal cache misses. As matrix sizes increase, they exceed the cache capacity, forcing the system to access slower main memory, which increases the execution time due to a higher cache miss ratio.

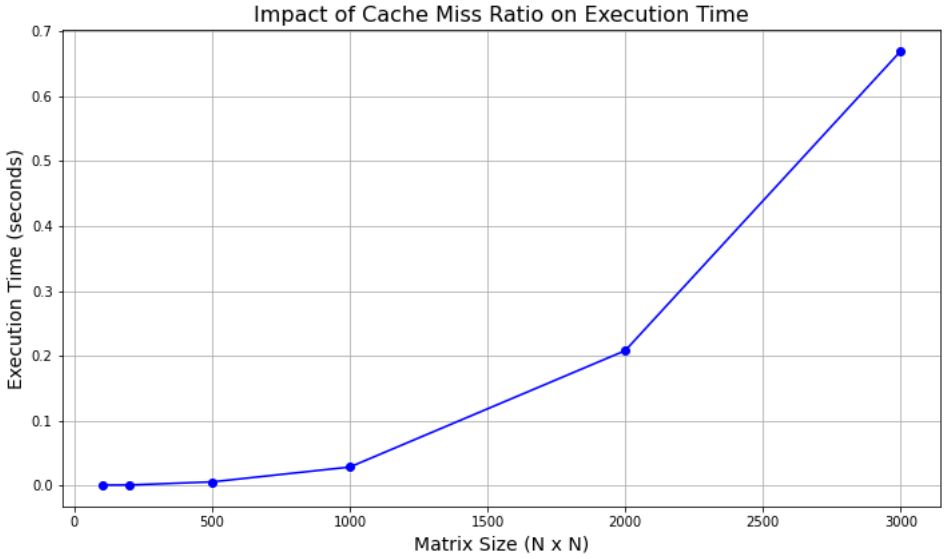


Figure 4: Impact of Cache Miss Ratio on Execution Time

The results clearly show that as matrix size increases, the execution time increases significantly. For smaller matrices like 100x100 and 200x200, the execution times are very short, reflecting efficient use of the CPU cache. However, as the matrix size increases to 500x500 and beyond, there is a steep increase in execution time. This trend continues with the 3000x3000 matrix, where the execution time is 0.669818 seconds, nearly 2000 times slower than for the smallest matrix.

These results highlight the significant impact of cache miss ratio on software performance. Smaller matrices are efficiently handled by the cache, leading to low execution times. However, as the matrices grow larger, they exceed the cache's capacity, causing more cache misses and forcing the system to access slower main memory. This shift leads to a dramatic increase in execution time, as evidenced by the nearly exponential growth in computation time for larger matrices. The experiment effectively demonstrates the critical role of the memory hierarchy, specifically cache utilization, in optimizing software performance for tasks like matrix multiplication.

5. the impact of TLB table miss ratio on the software speed performance (again, the software is supposed to execute relatively light computations such as multiplication)

   The goal of Experiment 5 was to measure how the TLB miss ratio affects software performance during matrix multiplication. The TLB stores page translations in memory, and a TLB miss occurs when a required page translation is not in the TLB, leading to more time-consuming page table walks. This experiment was designed to vary the TLB miss ratio by manipulating stride size during matrix multiplication. Larger strides increase the chance of TLB misses by spreading accesses across a larger range of memory pages.

   The Python code performs matrix multiplications across a range of matrix sizes and stride lengths to simulate varying TLB miss ratios. By adjusting stride size, we are simulating conditions that stress the TLB's capacity to cache page table entries. Larger strides are more likely to cause TLB misses since they lead to non-contiguous memory accesses.

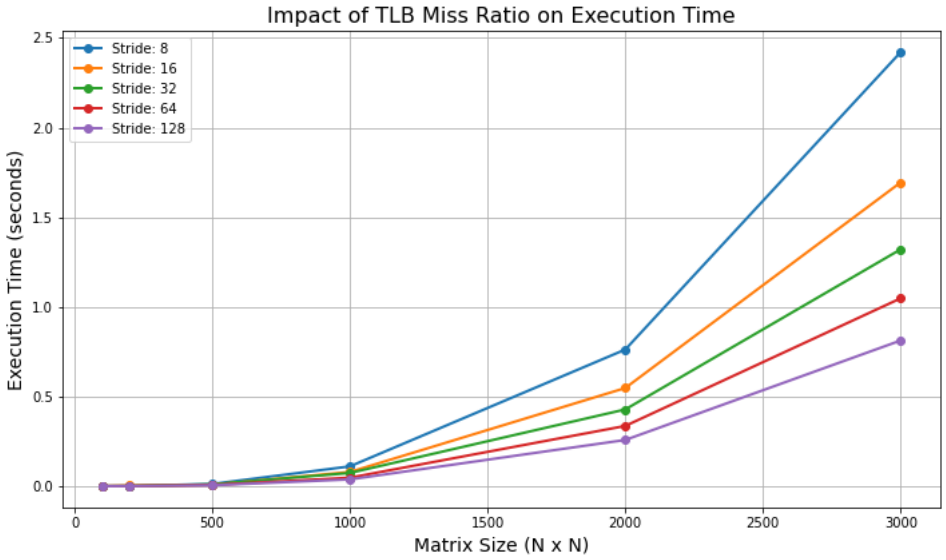


Figure 5: Impact of TLB Miss Ratio on Execution Time

The terminal results clearly show that both matrix size and stride size have a significant impact on execution time during matrix multiplication. For smaller matrices like 100x100, execution times are generally low, and the variation across different stride sizes is minimal. For instance, the execution time for a 100x100 matrix ranges from 0.000227 seconds at stride 8 to 0.000005 seconds at stride 128. However, as the matrix size increases, the differences in execution time become more pronounced. For a 3000x3000 matrix, execution time at stride 8 is 2.42 seconds, while at stride 128 it drops to 0.812 seconds. This pattern shows that larger strides, which reduce the number of memory pages accessed at once, lead to faster execution times. The graph reflects this trend as well, with larger matrices showing a sharp decrease in execution time as the stride increases, particularly in the transition from stride 8 to stride 128.

The results of this experiment confirm that TLB misses significantly affect performance, especially when dealing with larger datasets. Smaller strides, such as stride 8, cause memory to be accessed in a scattered manner across multiple pages, leading to more frequent TLB misses and higher execution times. In contrast, larger strides like stride 128, allow more contiguous memory access, reducing the number of TLB misses and significantly improving performance. This effect is most apparent in larger matrices like 2000x2000 and 3000x3000, where the overhead of TLB misses becomes more pronounced. The graph highlights this relationship, showing that execution time decreases as stride size increases, particularly for larger matrices. This experiment emphasizes the importance of optimizing memory access patterns to minimize TLB misses, which can have a substantial impact on software performance, especially in memory-intensive tasks like matrix multiplication.