### 0. Setup Methadology

The project was set up in a controlled Linux environment configured for performance benchmarking. A lightweight Ubuntu installation was used with background services minimized and CPU frequency fixed to performance mode to reduce variability.

Intel Memory Latency Checker (MLC) was employed to measure cache/memory latency and bandwidth. The perf tool collected hardware counters for cache and TLB analysis, enabling correlation with runtime through the AMAT model. Experiments were automated with scripts that ran multiple trials, logged results in CSV, and captured both latency and throughput.

Python post-processing filtered outliers, reporting variance, and plotting graphs. Benchmarks were pinned to dedicated cores with interference minimized, and results were validated against known cache boundaries and theoretical FLOP/bandwidth limits to ensure reproducibility.

**All code and scripts have been submitted (contain comments/greater detail on the specific implementation of each dataset).

1. **Zero-Queue Baselines:**

To isolate single-access latency at each level, we use Intel's MLC --idle_latency mode, which performs a pointer-chase (serial dependency) so that each memory access cannot begin until the previous one completes. This eliminates overlap and prefetching effects, yielding true "zero-queue" latency. By sweeping buffer sizes that fit in L1, L2, L3, and DRAM, we attribute the measured latencies to each hierarchy level.

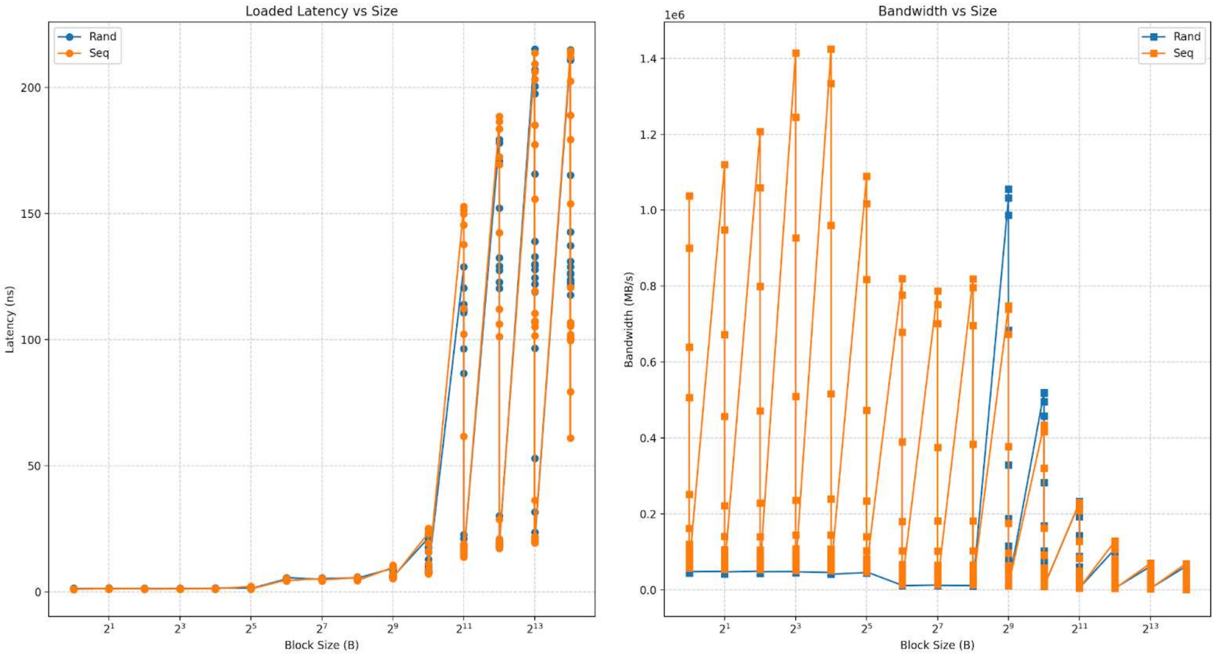
Using the below command, we can fill in the table
sudo ./mlc --idle_latency -bX where X = size in KB

**Table 1: Zero-queue baselines divided by memory hierarchies**

| | Size (KB) | cycles | Time (ns) | CPU Freq (GHz) |
|---|---:|---|---|---|
| L1 | 32 | 7.1 | 2.4 | 2.96 |
| L2 | 256 | 20.4 | 6.8 | 3.00 |
| LLC | 4000 | 79 | 26.4 | 2.99 |
| DRAM | 528000 | 460.5 | 153.7 | 2.99 |

2)

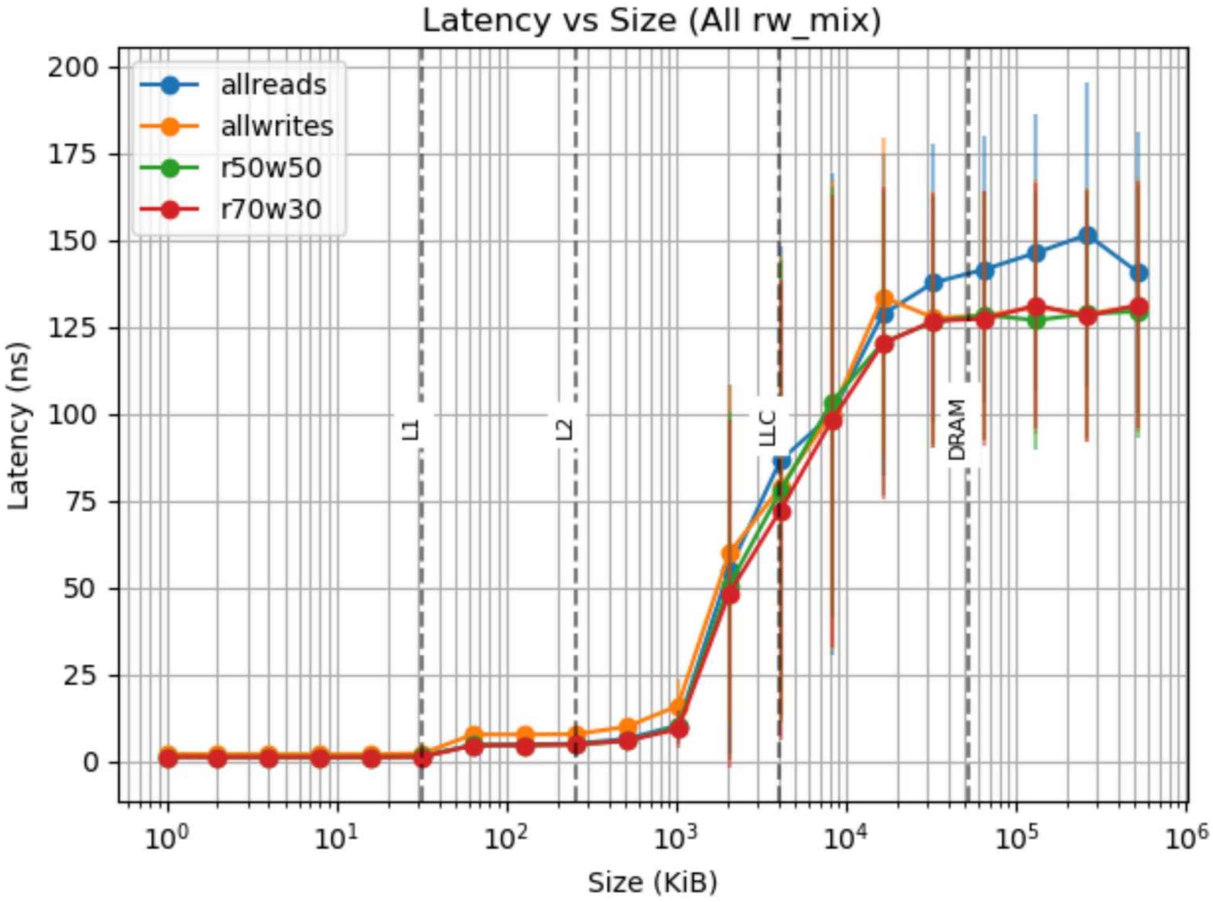
Latency Plot



Bandwidth Plot

The latency and bandwidth results reveal clear effects of cache, stride, and prefetching on memory performance. In the latency plot, both sequential and random accesses follow nearly identical trends, with a sharp increase around 2^10 bytes, corresponding to L1 cache saturation. The subsequent triangular-wave pattern in latency reflects periodic cache line conflicts and interactions with the hardware prefetcher, which can over- or under-fetch depending on stride alignment. For bandwidth, sequential accesses show higher throughput at small sizes due to effective prefetching, while random accesses are limited by lack of spatial locality. Around 2 9 2 9 bytes, the bandwidth of random accesses rises and converges with sequential accesses, indicating both patterns are now constrained by cache capacity and memory subsystem behavior. The triangular-wave decrease in sequential bandwidth illustrates stride-induced cache conflicts, where certain strides periodically underutilize cache lines. These results highlight the interplay of access pattern, stride, and prefetching in memory performance. Sequential accesses benefit from prefetching at small sizes, random accesses catch up once cache boundaries are exceeded, and the observed triangular-wave behavior underscores how stride and cache associativity can produce oscillatory latency and bandwidth patterns. Understanding these effects is essential for designing cache-efficient and high-throughput memory access patterns.

3)



The performance curves for all four mixes appear close together because they are governed by the same underlying **memory hierarchy** (L1, L2, last-level cache, and DRAM). At small sizes that fit within caches, latency is uniformly low and bandwidth extremely high regardless of mix, since the cache subsystem services both reads and writes efficiently. Due to the small data size, bandwidth is highly fluctuating as it is dependent on the specific block it loads. As the working set grows beyond cache capacity, all mixes converge to the sustained bandwidth and latency characteristics of DRAM, which dominate performance. Furthermore, the variance in the bandwidth decreases as more data is being loaded, reducing variance.

**Reads vs. Writes at Larger Sizes**

At small problem sizes, reads appear faster since they benefit from cache hits and low-latency paths through the CPU. As data grows beyond cache capacity, measured read latency rises because every access stalls until data returns from main memory. Writes, on the other hand, can be buffered and acknowledged early by the CPU, allowing the memory controller to drain them in the background. This buffering effect makes write latency appear lower than reads at large sizes, even though the physical memory operations take comparable time.

**All-Reads vs. Mixed Workloads**

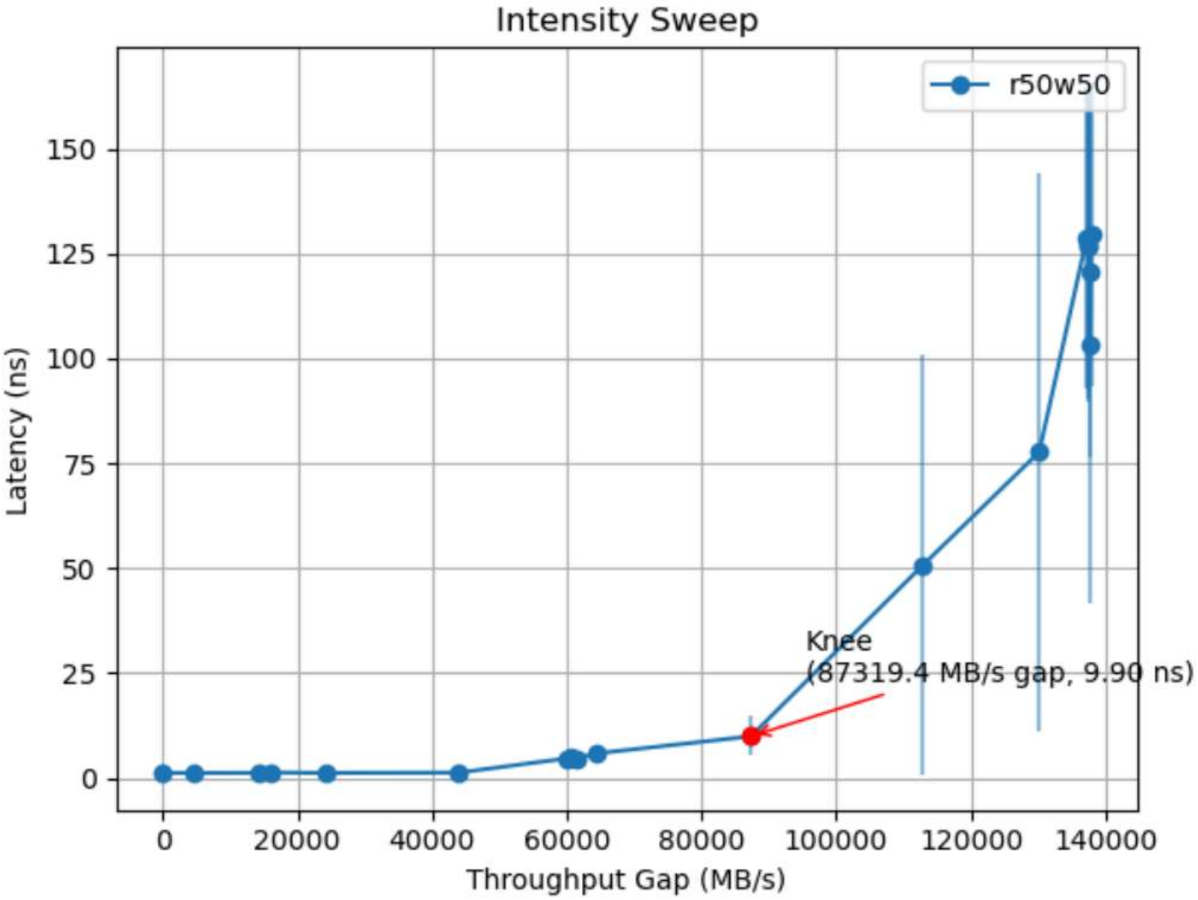
Pure read traffic often performs worse than a mix of reads and writes. This is because read-only access saturates the memory controller's read pipeline and leaves no opportunity to overlap or batch operations. When writes are introduced, the controller can drain them in bursts, freeing up banks and row buffers more efficiently. This leads to better pipelining and improved overall throughput, even though the workload seems more complex.

**Interpretation of Results**

These trends are expected and reflect system-level behavior rather than errors. Higher read latencies at scale arise from demand-fetch stalls, while buffered writes hide much of their cost. Similarly, mixed workloads expose optimizations in memory controllers that pure reads cannot exploit. Observing these effects demonstrates that latency and throughput depend not only on access size but also on workload composition, which is why both all-read and mixed patterns must be profiled for a complete picture.

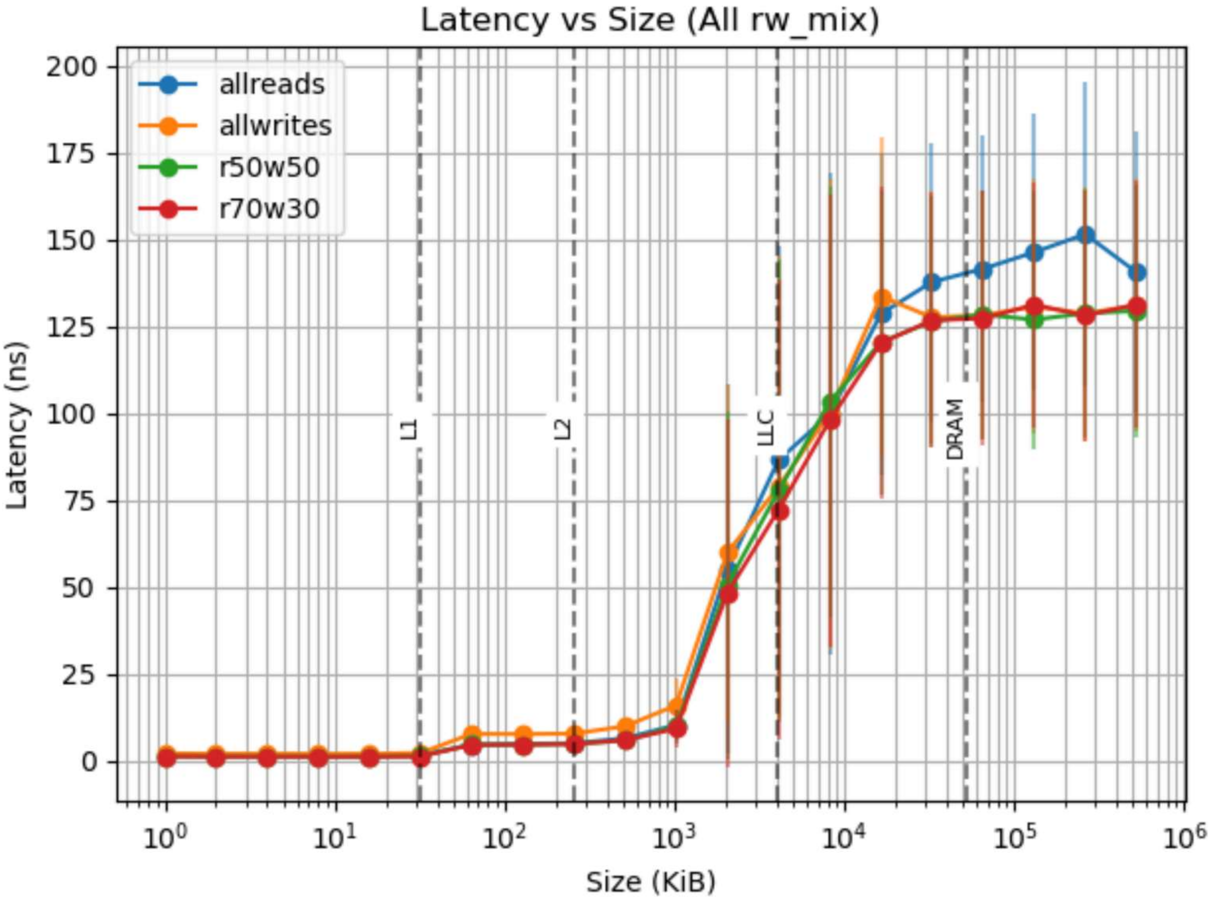
Overall, the closeness of the curves reflects the fact that the **system bottleneck is shared DRAM throughput**. Once accesses miss cache, memory becomes the limiting factor, and thus all mixes saturate at roughly the same latency and bandwidth plateaus, with only modest spread caused by how the memory subsystem handles reads versus writes.

4) Intensity Sweep



The knee is at 87.3 GB/s & 9.9 ns: pushing intensity beyond that point yields worse throughput and much higher latency, delivering 65% of peak bandwidth while still having latency and only having latency of 9.90 seconds (7.1% of peak latency) demonstrating classic diminishing returns predicted by Little’s Law.

5)



This graph clearly shows the latency as size increases. These values closely match the initial table.

6)

Cache-miss impact

Using cache calls continuously, and calling perf, we can find the time between each call. We get this data from using cache_misses.cpp, ./run.sh, and then ./strides.sh in the cache_tlb folder.

This benchmark shows how runtime per access changes as the working set size exceeds successive cache levels. When the footprint fits within L1 and L2 caches (≤256 KB), the average access cost is ≈0.31 ns, essentially flat. At 1 MB, the latency increases slightly to 0.41 ns, reflecting the onset of L2 capacity misses serviced from L3. Beyond this, the jump is more pronounced: at 4 MB the average latency rises to 1.04 ns, and by 16 MB it reaches 2.20 ns. These steps match the transition from L2 to L3 and then to main memory. Once the footprint exceeds the last-level cache (≥64 MB), the average access time stabilizes around ≈2.9 ns, indicating that virtually every access is a DRAM access.

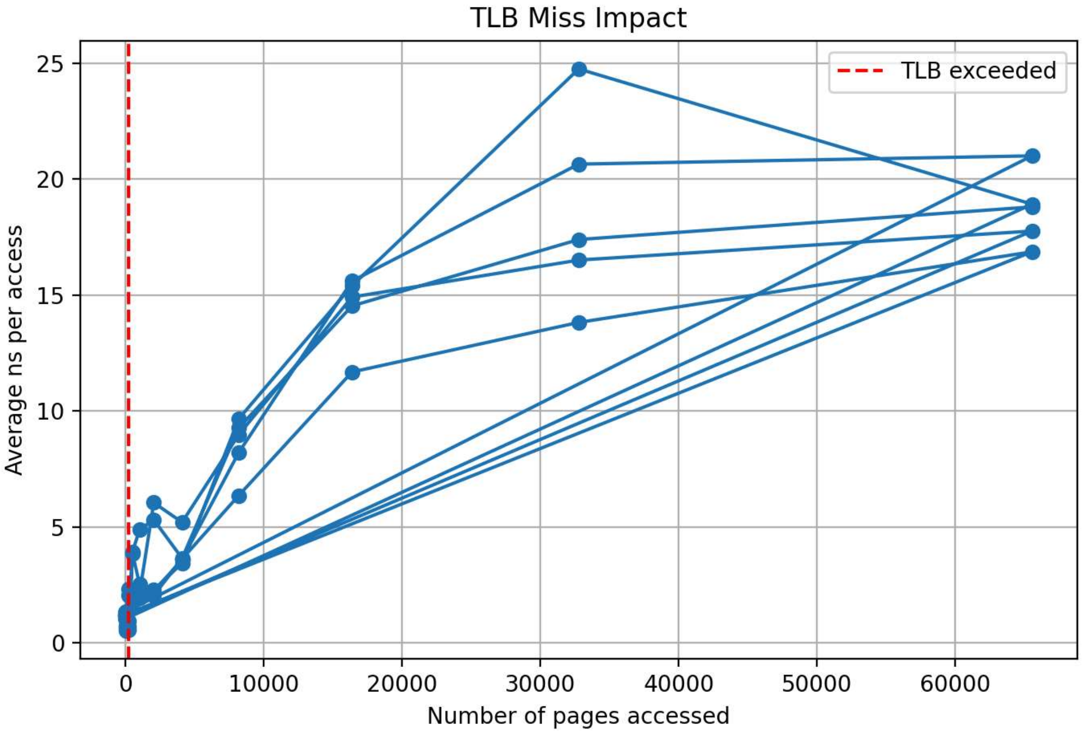
This increase in impact delay is explained in the AMAT model:
$AMAT=TL1+m1\cdot(TL2-TL1)+m2\cdot(TL3-TL2)+m3\cdot(TMem-TL3)$.

Even if we miss TL1, we still need to consider the impact time of it including the TL2 time. As the working set size increases, the probability of missing gets higher, increasing the latency.

| Footprint | Impact Time (ns) |
|---|---|
| 64 KB | 0.3 |
| 256 KB | 0.31 |
| 1 MB | 0.4 |
| 4 MB | 1 |
| 16 MB | 2.2 |

7)

TLB-miss impact



For small working sets up to around 128–256 pages, the measured latency per access remains low (0.5–1.3 ns) across all five runs. This indicates that the working set fits comfortably within the CPU's DTLB, and TLB misses are minimal. Latency values in this range are consistent, showing stable memory access without significant page translation overhead. As the working set increases beyond 512–1024 pages, latency begins to rise noticeably (1.8–2.5 ns), and by 2048 pages it jumps sharply (2–6 ns). Larger working sets, particularly 8192–65536 pages, show dramatic latency increases (6–20 ns), reflecting frequent TLB misses. These spikes occur because the number of pages accessed exceeds the DTLB reach, forcing the processor to perform page table walks for each uncached translation. Some fluctuations in intermediate values (256–1024 pages) likely result from caching interactions or slight variations in page mapping. The data also illustrates the theoretical benefit of huge pages: if each 2 MB page replaced the normal 4 KB pages, the effective DTLB reach would increase roughly 512×, reducing the number of TLB misses for the same working set. This would flatten the latency curve, eliminating the steep jumps observed for larger working sets. The results confirm that exceeding DTLB reach is the primary cause of the latency spikes seen in the benchmark, and that TLB-miss impact grows sharply with memory footprint once this hardware limit is exceeded.

****perf was not working due to ubuntu problems, and to approximate with my given data set and cpp, using other benchmarking tools such as MLC to aid in this. Therefore there is high variance in the graph.**