ACS Project 2 Report

Andrew Erickson

Introduction

Modern microprocessors have a well established memory hierarchy. Small amounts of high speed but expensive L1 cache sit closest to the processor core while larger, less expensive but slow DRAM sit some distance away on a circuit board. Because of this greater distance and different memory circuit topologies, DRAM typically has a high latency and the SRAM of the L1 cache has a very low latency. This report seeks to quantify these differences over a range of experiments.

Experimental set up

CPU: AMD Ryzen 5 PRO 4650U with Radeon Graphics
Clockspeed: 2.1 GHz
Turbo Speed: 4.0 GHz
Cores: 6
Threads: 12
Cache per CPU Package:

- L1 Instruction Cache: 6 x 32 KB
- L1 Data Cache: 6 x 32 KB
- L2 Cache: 6 x 512 KB
- L3 Cache: 8 MB

DRAM: 16 GB (2 x 8 GB) DDR4-3200 OS: Win 10
Compiler: g++, version: 13.2.0
IDE Used: VSCode

Section 1: Zero-queue baselines

CPU Frequency: 3.7GHz

Memory Level	Size	Latency (ns)	Latency (Cycles)	Relative to L1
L1	32.0 KB	1.1 ns	4.07	1.0x
L2	512.0 KB	4.9 ns	18.13	4.5x

L3	8.0 MB	107.3 ns	397.01	97.5x
DRAM	64.0 MB	116.8 ns	432.16	106.2x

This chart shows the memory hierarchy of a CPU system, illustrating the classic trade-off between size and speed. As you move down the hierarchy from L1 cache to DRAM, storage capacity increases dramatically (from 32 KB to 64 MB), but so does access latency. L1 cache is the fastest at just 4 cycles, while DRAM is over 100x slower at 432 cycles. The L2 cache provides a middle ground at 4.5x L1 latency, but L3 cache shows a significant jump to nearly 100x slower than L1, approaching DRAM speeds. This demonstrates why keeping frequently accessed data in faster, smaller caches is critical for performance.

This data and the Intel MLC commands used to produce it are in the `base data` folder

Section 2: Pattern & Granularity Sweep

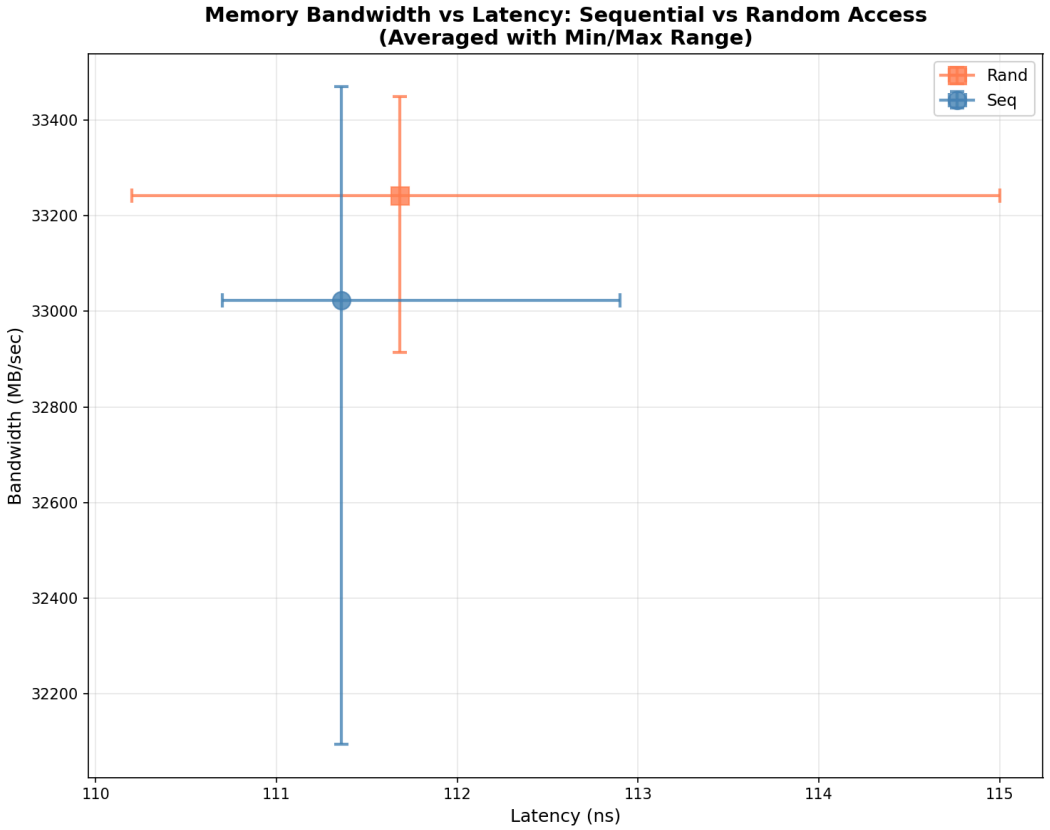

Figure 1: Sequential vs Random Access Pattern

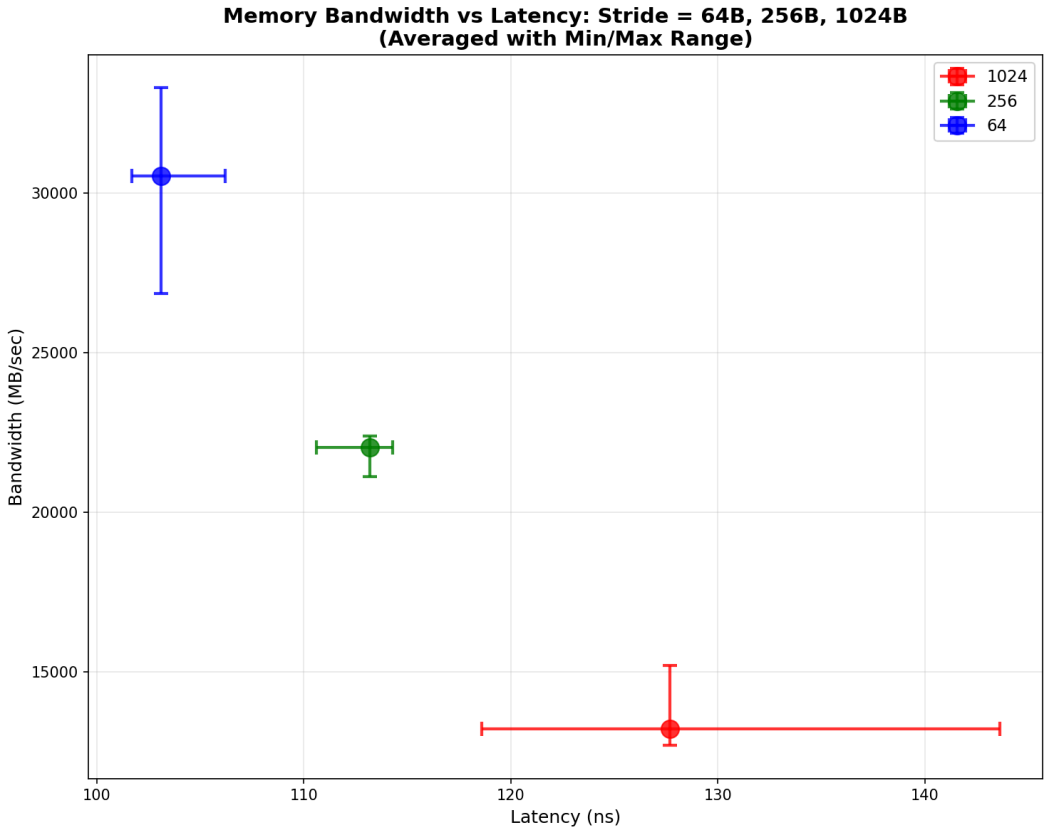

Figure 2: Differing Stride Access Pattern

These two figures reveal critical insights into how modern memory systems handle different access patterns and the effectiveness of hardware prefetching mechanisms.

Sequential vs Random Access: Figure 1 demonstrates that sequential and random access patterns achieve nearly identical bandwidth (~33,000 MB/sec) at similar latencies (~111-115 ns). This is counterintuitive at first glance, as random access typically performs worse. However, the similar performance suggests that both access patterns are likely bottlenecked by DRAM bandwidth and not latency. The tighter error bars on the Sequential access indicate more consistent performance, meaning the memory controller is efficiently saturating the available memory bandwidth. This feat is harder to accomplish with the Random access. The latency values (~111-115 ns) align closely with your DRAM latency measurement (116.8 ns), confirming these are DRAM-bound operations.

Stride Effects: Figure 2 reveals dramatic performance differences based on stride size, clearly illustrating the impact of hardware prefetching:

- 64B stride (cache line size): Achieves the highest bandwidth (~30,000 MB/sec) with lowest latency (~105 ns). This represents optimal sequential access where hardware prefetchers can predict the pattern perfectly. Each access naturally loads the next cache line, and prefetchers stay ahead of demand.

- 256B stride: Shows moderate performance (~22,000 MB/sec, ~115 ns latency). The prefetcher can still detect and exploit this regular pattern, though with reduced effectiveness. The larger gaps between accesses reduce spatial locality benefits.
- 1024B stride: Exhibits severe performance degradation (~13,000 MB/sec, ~127 ns). This stride is too large for effective prefetching as DRAM latency dominates. The memory system essentially treats this as random access, missing most prefetch opportunities. Bandwidth drops to less than half of the 64B case, demonstrating how critical prefetching is to modern memory performance.

Key Takeaways The results emphasize that modern memory hierarchies are highly optimized for sequential access through sophisticated prefetching hardware. Applications should structure data access to maintain spatial locality within cache lines (64B) whenever possible. Large strides effectively disable prefetching benefits, causing performance to collapse even when the access pattern is technically regular and predictable.

Section 3: Read / Write Mix Sweep

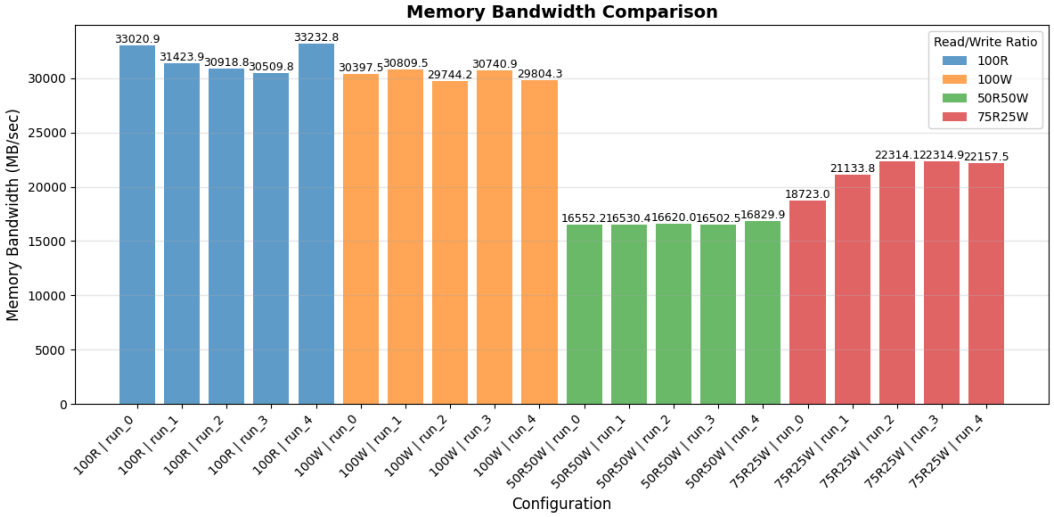

Figure 3: Differing Stride Access Pattern

Figure 3 reveals how memory bandwidth performance degrades systematically as the proportion of write operations increases, demonstrating fundamental asymmetries in modern memory architectures.

The results show a clear three-tier performance structure:

100% Read (100R) - Highest Performance (~30,000-33,000 MB/sec)

- Pure read workloads achieve the maximum bandwidth with descent consistency across all five runs. This represents the optimal case for several reasons. Read operations can be heavily pipelined and buffered by the memory controller, multiple outstanding read requests can be in flight simultaneously, and read data can be speculatively prefetched. The memory system is architecturally optimized for read-heavy workloads since most applications exhibit this pattern.

100% Write (100W) - Slightly Reduced Performance (~29,000-30,800 MB/sec)

- Write-only operations show approximately 3-5% lower bandwidth compared to reads. While this might seem surprising, writes can still achieve high throughput through write combining buffers and posted writes that don't require immediate completion acknowledgment. The memory controller can batch multiple writes together and optimize their execution. However, writes cannot benefit from prefetching and require more careful ordering to maintain data consistency, explaining the slight performance reduction.

Mixed Workloads - Progressive Degradation

- The 50R50W (50% reads, 50% writes) configuration shows significant bandwidth reduction to ~16,500 MB/sec, representing roughly 50% of the pure read bandwidth. The 75R25W ratio (75% reads, 25% writes) performs at ~18,700-22,300 MB/sec, falling between the mixed and read-heavy extremes. These dramatic drops reveal that mixing reads and writes creates fundamental inefficiencies in the memory subsystem.

Several architectural factors contribute to the poor performance of mixed read/write workloads:

1. Memory Controller Scheduling Conflicts: The memory controller must switch between read and write modes. DRAM banks require different timing parameters for reads versus writes, and transitioning between modes introduces idle cycles. Read and write queues compete for the same memory channels, creating scheduling conflicts that reduce overall throughput.
2. Write Buffer Saturation: While writes can be buffered temporarily, a continuous stream of mixed operations prevents the write buffer from draining efficiently. The memory controller must periodically flush write buffers to maintain consistency, which blocks read operations and creates pipeline stalls.
3. Reduced Prefetching Effectiveness: Write operations disrupt the regular access patterns that hardware prefetchers rely on. Prefetchers are optimized for read-ahead behavior, but writes don't benefit from prefetching and can invalidate prefetched data. Mixed patterns make it harder for the prefetcher to predict future accesses accurately.
4. Cache Coherency Overhead: In systems with multiple cores or memory-mapped I/O, writes trigger additional coherency traffic. The memory system must ensure all cached copies are invalidated or updated, adding latency that doesn't exist for pure reads.

Key Insights

The roughly 50% bandwidth reduction for 50R50W workloads demonstrates that the performance penalty isn't simply additive. The interference between reads and writes creates superlinear degradation, where the combined workload performs worse than the weighted average of pure read and write bandwidths. This highlights why performance-critical applications often separate read and write phases or use separate buffers to minimize mixed-access patterns. Understanding these bottlenecks is crucial for optimizing memory-intensive algorithms and data structure design.

We could also try, in our algo design, to batch many reads/writes at the same time in the algo in order to take advantage of our memory hardware which clearly

prefers to exclusively read or exclusively write large amounts of data instead of constantly switching modes.

Section 4: Intensity Sweep

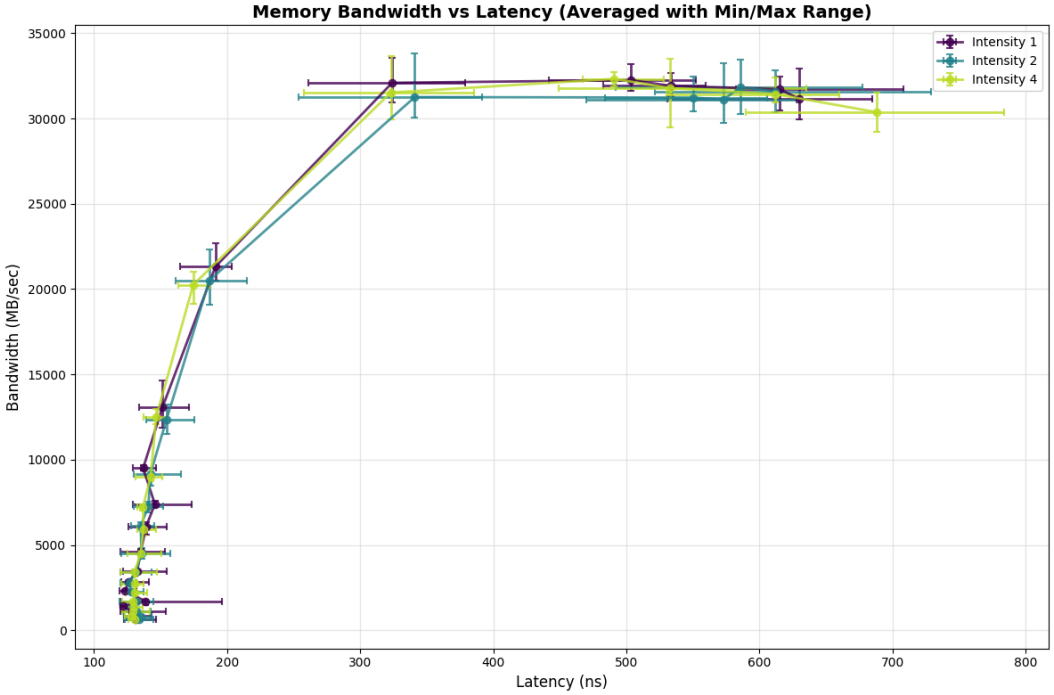

Figure 4: Bandwidth vs Latency with Differing Intensity

Figure 4 demonstrates the relationship between memory access latency and achieved bandwidth across three different access intensities, revealing critical insights about memory system saturation and parallelism requirements.

The performance curve exhibits a sharp "knee" at approximately 200-300 ns latency. Below this point, bandwidth scales nearly linearly with latency across all intensity levels, climbing steeply from under 10,000 MB/sec to over 20,000 MB/sec. Beyond 300 ns, the curves flatten dramatically and plateau at roughly 31,000-32,000 MB/sec, indicating saturation of the memory subsystem. This knee represents the transition from a latency-limited regime (insufficient in-flight requests) to a bandwidth-limited regime (memory controller fully saturated).

Assuming a theoretical peak of approximately 33,000 MB/sec (based on the asymptotic behavior), the system achieves 94-97% of theoretical peak once past the knee. The key observation is that beyond 300 ns latency, adding more outstanding requests yields minimal improvement—less than 3% additional bandwidth despite doubling or tripling the latency. This demonstrates severe diminishing returns: the first 200 ns of latency provides 60-65% of peak bandwidth, while the next 500+ ns adds only ~30% more. Performance-critical applications should target the knee region to balance bandwidth utilization against the overhead of managing excessive outstanding requests.

These results directly validate Little's Law: Throughput = Concurrency / Latency. To maintain constant throughput (bandwidth) as latency increases, concurrency

(number of outstanding memory requests) must increase proportionally. The different intensity levels represent varying degrees of concurrency. At low latencies (100-150 ns), even low-intensity workloads achieve decent bandwidth because few outstanding requests are needed. However, as latency extends to 300+ ns, higher intensity (more concurrent requests) becomes essential to saturate the memory pipeline. The plateau indicates we've reached the maximum sustainable throughput; adding more concurrency cannot overcome the fundamental bandwidth limit of the memory channels themselves.

Section 5: Working-set Size Sweep

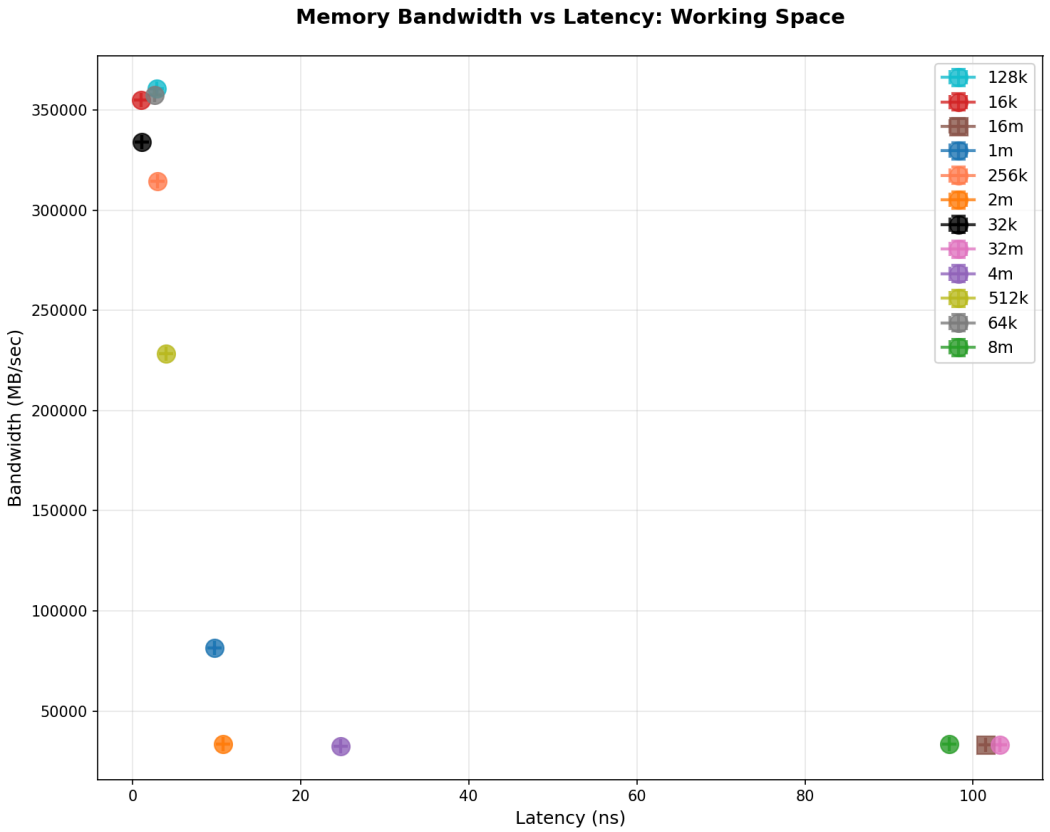

Figure 5: Bandwidth vs Latency with Differing Working Space

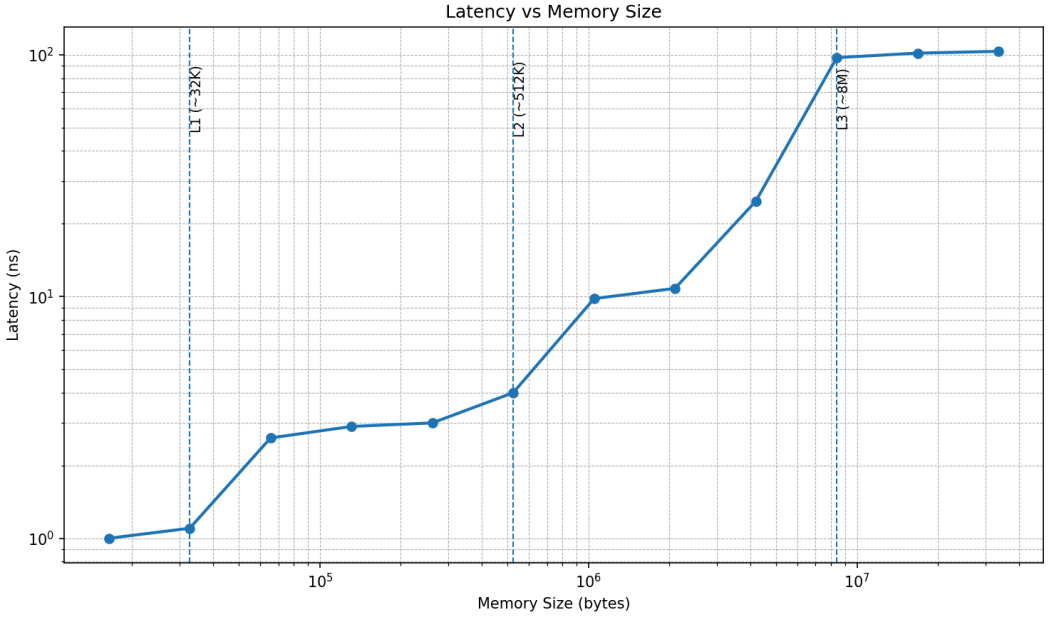

Figure 6: Memory Size vs Latency with Cache Transitions

These two graphs illustrate how working set size dramatically affects both memory latency and bandwidth, revealing the hierarchical nature of the cache system.

Figure 5 reveals severe performance penalties as working sets exceed cache capacities. Small working sets (16K-128K) achieve peak bandwidth of ~35,000 MB/sec with extremely low latency (<5 ns), indicating they remain cache-resident. Mid-size working sets (256K-1MB) show moderate bandwidth (~30,000-80,000 MB/sec) with latencies of 5-15 ns as they begin utilizing L2/L3 caches. Large working sets (2M-32M) that exceed L3 capacity suffer catastrophic bandwidth collapse to ~30,000 MB/sec with latencies jumping to 25-105 ns, as they're forced to access DRAM repeatedly.

Figure 6 shows three distinct plateaus corresponding to different cache levels. Working sets under ~32KB fit entirely in L1 cache with minimal latency (~1 ns). Between 32KB and ~512KB, data resides in L2 cache with latency around 3-4 ns. The steep climb from 512KB to ~8MB indicates L3 cache access with latencies increasing from 4 ns to approximately 10 ns. Beyond 8MB, the working set spills to DRAM, causing latency to jump dramatically to ~100 ns—a 10x increase. The vertical dashed lines marking L1, L2, and L3 boundaries clearly show these capacity thresholds align with the performance cliffs.

The critical takeaway is the sharp performance cliff when working sets exceed the largest cache level (L3 at ~8MB). Applications should strive to keep hot data within L3 capacity at most, as crossing into DRAM results in both 10x higher latency and dramatically reduced effective bandwidth due to cache thrashing and increased miss rates.

Section 6: Cache Miss Impact

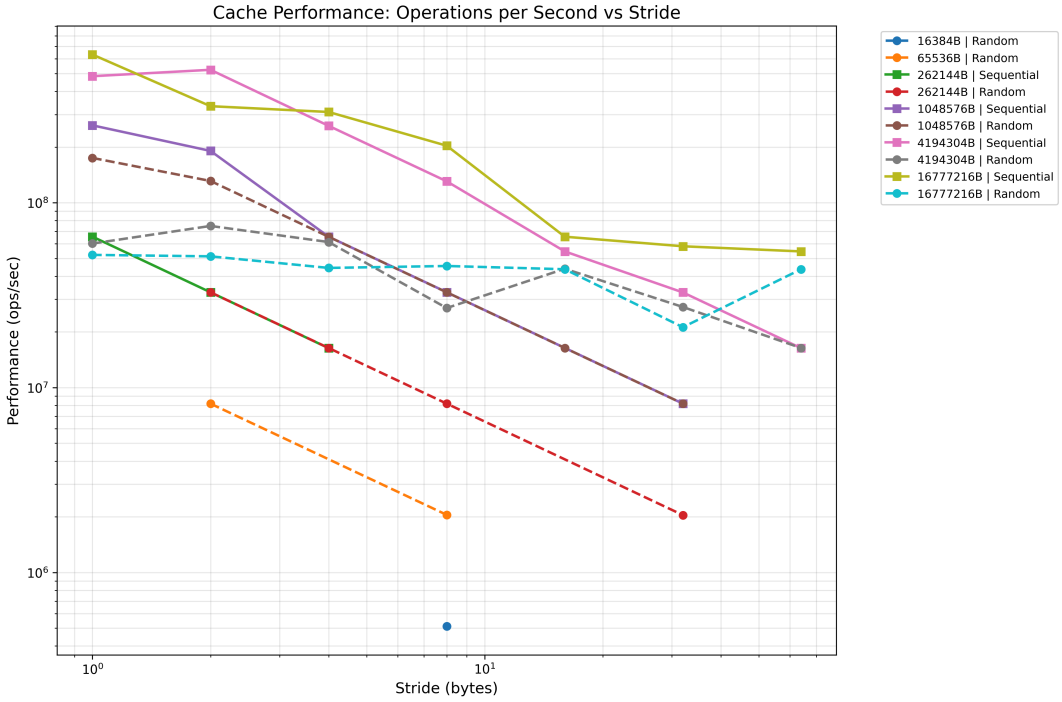

Figure 7: Performace vs Stride with Changing Access Pattern

Figure 7 demonstrates how stride and access patterns affect cache performance across different working set sizes, with results directly explainable through the Average Memory Access Time (AMAT) model: AMAT = Hit Time + (Miss Rate × Miss Penalty).

Sequential access (Solid Lines) maintains consistently high performance across all stride values, with the largest working sets (16777216B and 4194304B) achieving ~10^8 to ~10^9 ops/sec. This behavior reflects the AMAT model where hardware prefetchers dramatically reduce the effective miss rate. Even when the working set exceeds cache capacity, sequential prefetching keeps the next cache line loaded before it's needed, maintaining a low miss rate despite the large dataset. The slight performance degradation at larger strides (stride ≥ 4) occurs because prefetching becomes less effective when stride exceeds the prefetch distance, slightly increasing the miss rate component of AMAT.

Random access (Dashed Lines) shows catastrophic performance collapse, particularly for larger working sets. The 262144B random access drops from ~10^7 ops/sec at stride 1 to ~10^6 ops/sec at stride 8, representing a 10x performance loss. This directly reflects AMAT: random patterns eliminate prefetcher benefits, creating near-100% miss rates for working sets exceeding cache capacity. Each miss incurs the full miss penalty (L3 or DRAM latency), dominating the AMAT calculation. Smaller working sets like 16384B maintain better performance because they fit in L1/L2 cache, keeping the miss rate low and AMAT dominated by the fast hit time rather than miss penalty.

The vertical separation between curves of different sizes quantifies AMAT's miss penalty term. Small working sets (16384B, 65536B) stay cache-resident with high hit rates, resulting in AMAT ≈ Hit Time (~1-4 ns), yielding ~10^8 ops/sec. Large working sets (1048576B+) that exceed L3 capacity experience high miss rates to DRAM, where AMAT ≈ Miss Rate × Miss Penalty ≈ 1.0 × 100ns = 100ns, limiting performance to ~10^7 ops/sec or worse. The stark 10-100x performance difference between small and large random-access working sets directly demonstrates how the miss penalty term dominates AMAT when data doesn't fit in cache.

Figure 7 validates that optimizing cache performance requires minimizing both miss rate (through sequential access and appropriate working set sizing) and miss penalty (by keeping data in faster cache levels). Random access with large working sets maximizes both terms, creating the worst-case AMAT scenario visible in the bottom curves.

Section 7: TLB Miss Impact

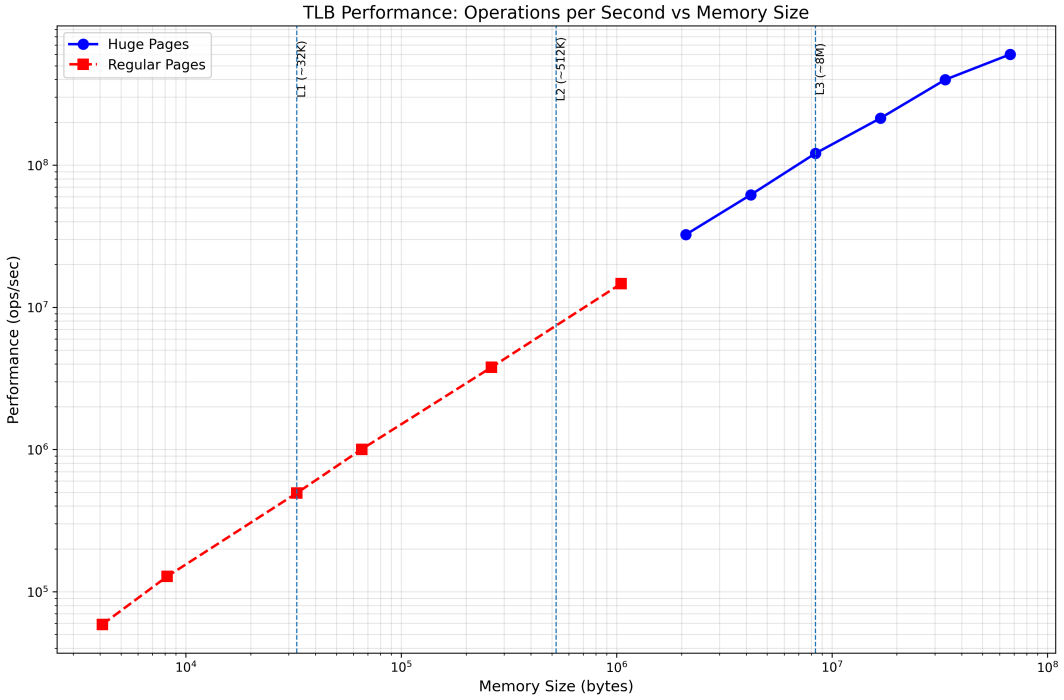

Figure 8: Performace vs Stride with Changing Access Pattern

Figure 8 demonstrates the dramatic performance advantage of huge pages over regular pages, directly illustrating how TLB reach limitations create scalability bottlenecks for memory-intensive applications.

The experiment methodically varies memory footprint from ~8KB to ~1GB while comparing 4KB regular pages against 2MB huge pages. Regular pages (red dashed line) show continuous performance degradation as memory size increases, dropping from ~10^5 ops/sec at 8KB to ~10^7 ops/sec at 1MB, before plateauing. In stark contrast, huge pages (blue solid line) maintain consistently high performance of ~10^8 ops/sec across the entire range, showing minimal degradation even at 1GB. This 10-100x performance difference directly quantifies the cost of TLB thrashing with regular pages.

The vertical dashed lines mark critical TLB coverage boundaries. L1 DTLB (~32KB) represents the first-level data TLB reach with 4KB pages (typically 64 entries × 4KB = 256KB, though the graph shows ~32KB suggesting a smaller configuration). Beyond this point, regular pages experience their first performance drop as L1 DTLB misses require L2 TLB lookups. L2 DTLB (~512KB) marks the combined L1+L2 TLB reach. Once exceeded, every memory access risks a costly page table walk, causing the severe performance degradation visible in the regular page curve between 512KB and 8MB. L3 DTLB (~8MB) likely represents an approximate effective reach including some page walk cache benefits. Beyond 8MB, regular pages plateau at their worst-case performance dominated by continuous page table walks to DRAM.

The DTLB reach—the maximum memory addressable through TLB entries without page walks—fundamentally determines performance scalability. With 4KB pages and typical DTLB configurations (64 L1 entries + 1024 L2 entries), reach is limited to approximately 4MB total (4KB × 1088 entries). The regular page curve shows exponential performance collapse precisely as working sets exceed this reach, confirming TLB misses dominate execution time. Each TLB miss incurs a page table walk requiring multiple memory accesses (typically 4 levels for x86-64), adding hundreds of cycles per miss.

Huge pages (2MB) multiply DTLB reach by 512x per entry, extending coverage to ~2GB with the same TLB hardware. This explains why huge pages maintain flat, optimal performance across the entire test range—the working set never exceeds TLB reach, eliminating page walks entirely. The performance gap widens dramatically as memory footprint grows: at 1GB, huge pages deliver ~10^8 ops/sec while regular pages struggle at ~10^7 ops/sec, representing a 10x throughput advantage solely from eliminating TLB misses. This demonstrates that for workloads with large memory footprints, TLB reach becomes the dominant performance bottleneck, making huge pages essential for achieving near-optimal memory system performance.

Conclusion

This report systematically explored the performance characteristics of modern memory hierarchies through comprehensive empirical analysis. The experiments quantified fundamental trade-offs between capacity, latency, and bandwidth across multiple levels of the memory system, from L1 cache to DRAM.

Several critical findings emerged from this investigation. First, the memory hierarchy exhibits stark performance cliffs at cache capacity boundaries, with L3-to-DRAM transitions showing 10x latency increases and corresponding bandwidth degradation. Second, hardware prefetching mechanisms prove essential for maintaining performance, as demonstrated by the dramatic differences between sequential and random access patterns—sequential access sustains near-peak bandwidth even with large working sets, while random access suffers order-of-magnitude performance losses. Third, mixed read/write workloads exhibit superlinear performance degradation due to memory controller scheduling conflicts and cache coherency overhead, achieving only 50% of pure read bandwidth at 50/50 ratios.

The intensity sweep validated Little's Law, confirming that sufficient memory-level parallelism (300+ ns worth of outstanding requests) is necessary to saturate modern DRAM bandwidth. However, diminishing returns beyond this saturation point suggest that simply increasing concurrency provides minimal benefit once the memory controller reaches its throughput limit.

Perhaps most significantly, the TLB analysis revealed that virtual memory translation overhead can become the dominant bottleneck for large working sets. Regular 4KB pages suffer 10-100x performance degradation beyond 4MB working sets due to page table walk costs, while 2MB huge pages maintain optimal performance across the entire tested range by extending TLB reach from ~4MB to ~2GB.

These results underscore that achieving high memory system performance requires holistic optimization: structuring data access for spatial locality to leverage prefetching, sizing working sets to fit within cache hierarchies, batching homogeneous operations to avoid read/write mixing penalties, maintaining sufficient memory-level parallelism to saturate bandwidth, and deploying huge pages for large memory footprints. Understanding these architectural characteristics enables developers to make informed design decisions that align algorithmic behavior with hardware capabilities, ultimately extracting maximum performance from the memory subsystem.