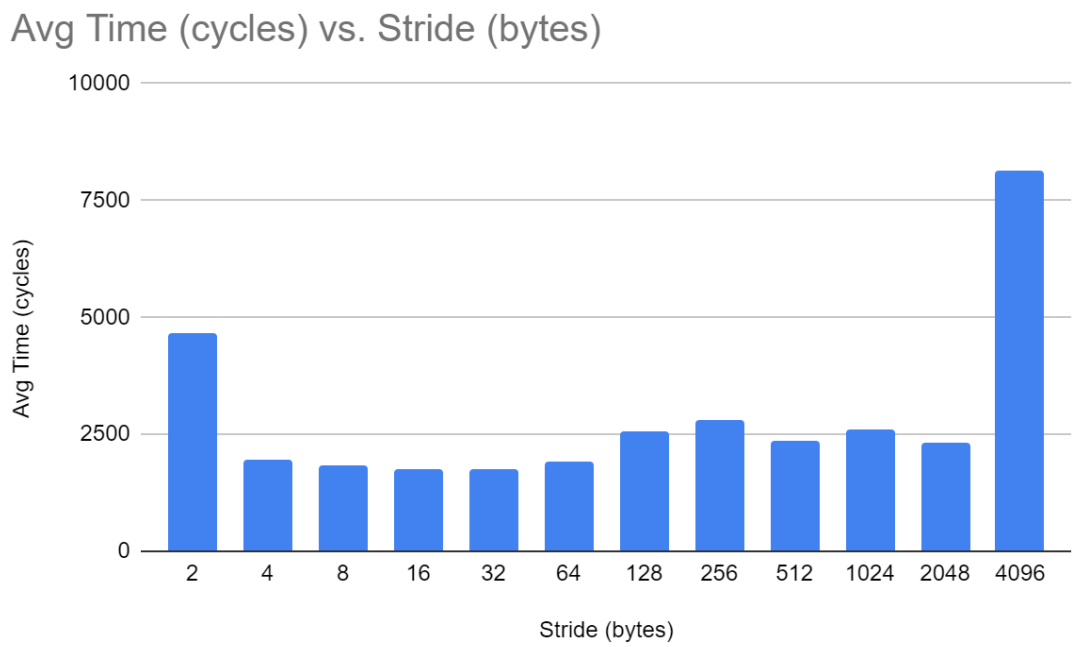
Note: The tests were measured using both `rdtsc` and `timeval` values. `rdtsc` was preferred due to its higher granularity compared to `timeval`, providing more precise measurements of access times.

Cache Line Size

Assumptions:
In typical systems, the cache line size is commonly 64 bytes.

Methodology:
To investigate cache line effects, we use `aligned_malloc` to allocate memory with a 64-byte alignment, facilitating accurate measurement of stride access performance. To enforce sequential execution of loads we employ dependency hazards such as read after write. To avoid prefetching effects, we employ a loop with varying access addresses creating random access noise and minimizing prefetching effects. During the test, we measure access times for each stride size and average the results over 100 iterations. We anticipate that access times will increase noticeably starting from a stride size of 128 bytes, as this involves fetching additional cache lines.

Results:
The data indicates a high number of cycles for the smallest stride size due to the initial cache line fetch. Access times for strides between 2 and 64 bytes are relatively low. However, as stride sizes increase, particularly beyond 64 bytes, there is a noticeable rise in cycle counts, reflecting the increased need to fetch additional cache lines. The substantial peak observed at a stride of 4096 bytes likely results from cache misses due to the exceedingly large stride size.

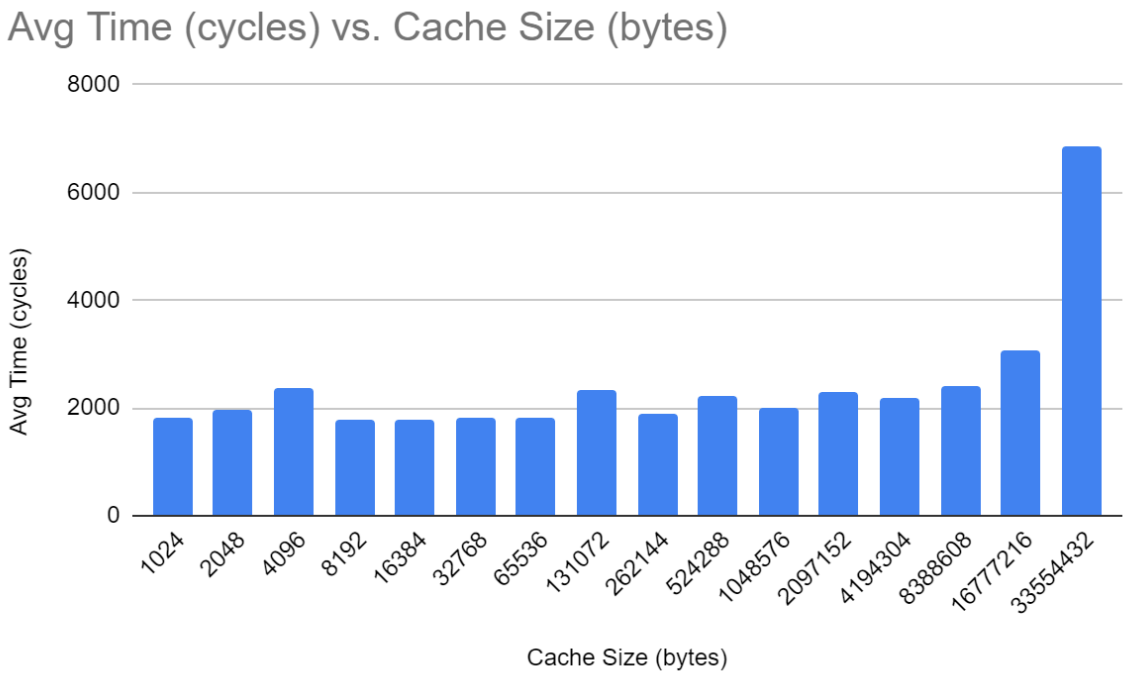


Capacity of Last-Level Cache

Assumptions:
The test is conducted on a system with a Zen 4 architecture, specifically the Ryzen 7 8845HS CPU. Typical cache sizes for this architecture are L1: 512 KB, L2: 8 MB, and L3: 16 MB.

Methodology:
The `CacheSizeTest` function evaluates the impact of various cache sizes on memory access times. The function tests cache sizes ranging from 1 KB to 32 MB by first flushing the cache with `flush_cache` to ensure a clean slate. It then allocates memory of the current cache size using `aligned_allocate`, initializes it to simulate cache filling, and measures access times using both CPU cycles (via `rdtsc`) and wall-clock time (via `gettimeofday`). This process involves randomly accessing a memory location within the allocated array and calculating average access times over 1000 runs for each cache size. Results are averaged and reported to show how different cache sizes influence performance.

Results:
The results align with expectations for the L2 cache size of 8 MB, as access times are consistently around 2000 cycles. When the cache size is increased to 16 MB, we observe a rise in access times to approximately 4000 cycles. At a cache size of 32 MB, the number of cycles increases significantly to 6865, indicating that accesses are reaching main memory and not being served by the CPU caches.



Memory Latency to DRAM

Assumptions:
We assume that the L3 cache size is 16 MB.

Methodology:
To measure the latency to DRAM, the test leverages `clflush/mfence` to clear the cache in 64-byte increments, corresponding to the cache line size. The procedure begins with a cache flush, followed by populating the cache by accessing an `aligned_malloc` array. The test then measures the access time for a memory location before and after a second cache flush. Specifically, the function `MemoryTimingTest(size_t array_size)` is executed over 1000 runs. Within each run, the following steps are performed:

1. **Cache Warm-Up**: The cache is initially warmed up by accessing the array in 64-byte strides.
2. **Initial Measurement**: Access time for a single memory location is recorded before the cache is flushed.
3. **Cache Flush**:
    a. Iterate over the allocated memory in steps of `cl` (cache line size) and issue a `clflush` instruction for each cache line. The `clflush` instruction is used to invalidate and flush a specific cache line from the CPU cache.
    b. The `mfence` instruction is used to ensure that all previous cache line flushes are completed before proceeding..
4. **Secondary Measurement**: Access time for the same memory location is measured again after the cache flush.

Results:
Initial measurements show an access time of 0.000020 ms before the cache flush and 0.000060 ms after the flush (20 ns , 60 ns respectively). These results are consistent with the expected latency for DDR5 memory, reflecting the impact of cache flushes on access times. The significant increase in access time after flushing indicates the transition from cache-resident to DRAM-resident data, highlighting the latency incurred when data is no longer available in the cache and must be fetched from DRAM.

Associativity

Assumptions:
We assume that the Last-Level Cache (LLC) is 16-way set associative, with a cache size of 16 MB and a cache line size of 64 bytes.

Methodology:
To determine the number of cache ways, we access an array with increments surrounding 1,048,576 (2^20), which includes both the set and offset index. This number was found through our assumptions of way size, line size and cache size. Cache Size / (Line Size * Way Size) = Number of Sets -> log2(Number of Sets) = Number of Bits used to index Sett. Total Bits = Set Bits + Offset Bits. By indexing through the array at these increments, we continuously access a specific set within the cache. This process leads to multiple cache misses, which results in a higher number of cycles required to pass through the test.

Breakdown of `CacheAssocTest` function:

1. **Setup**: Allocate an array with a size specified by `array_size`, ensuring proper alignment with a 64-byte boundary.
2. **Cache Warm-Up**: Iterate over the array in 64-byte increments to ensure that the cache is populated, which helps in establishing a baseline.
3. **Test Execution**:
    - **Cache Flushing**: Before each set of measurements, flush the cache to clear any existing data.
    - **Stride Access Testing**: For a range of stride sizes (from 1,048,500 to 1,048,600 bytes), measure the access times. This range is chosen to investigate the behavior of the cache around the expected number of cacheways.
    - **Timing Measurements**: For each stride size, access the array at intervals of the current stride size. Record access times using both `rdtsc` for cycle counts and `timeval` for time in milliseconds. The `rdtsc` instruction provides high-resolution timing, while `timeval` is used for broader time measurement.
4. **Calculation**:
    - Compute the total cycles and time values for each stride size.
    - Average these measurements over the number of runs and accesses to obtain the average cycles and time per access for each stride size.
5. **Results**:
    - Print the average access times in cycles and milliseconds for each stride size.
    - Analyze the results to identify the stride size at which the number of cycles increases significantly, indicating the point at which cache misses become more frequent due to cache set associativity.

Results:
The results show some noise in the data, but there is a clear peak around the 1,048,576 mark. This peak indicates the point at which cache misses become more frequent, confirming the expected 16-way associativity of the cache.