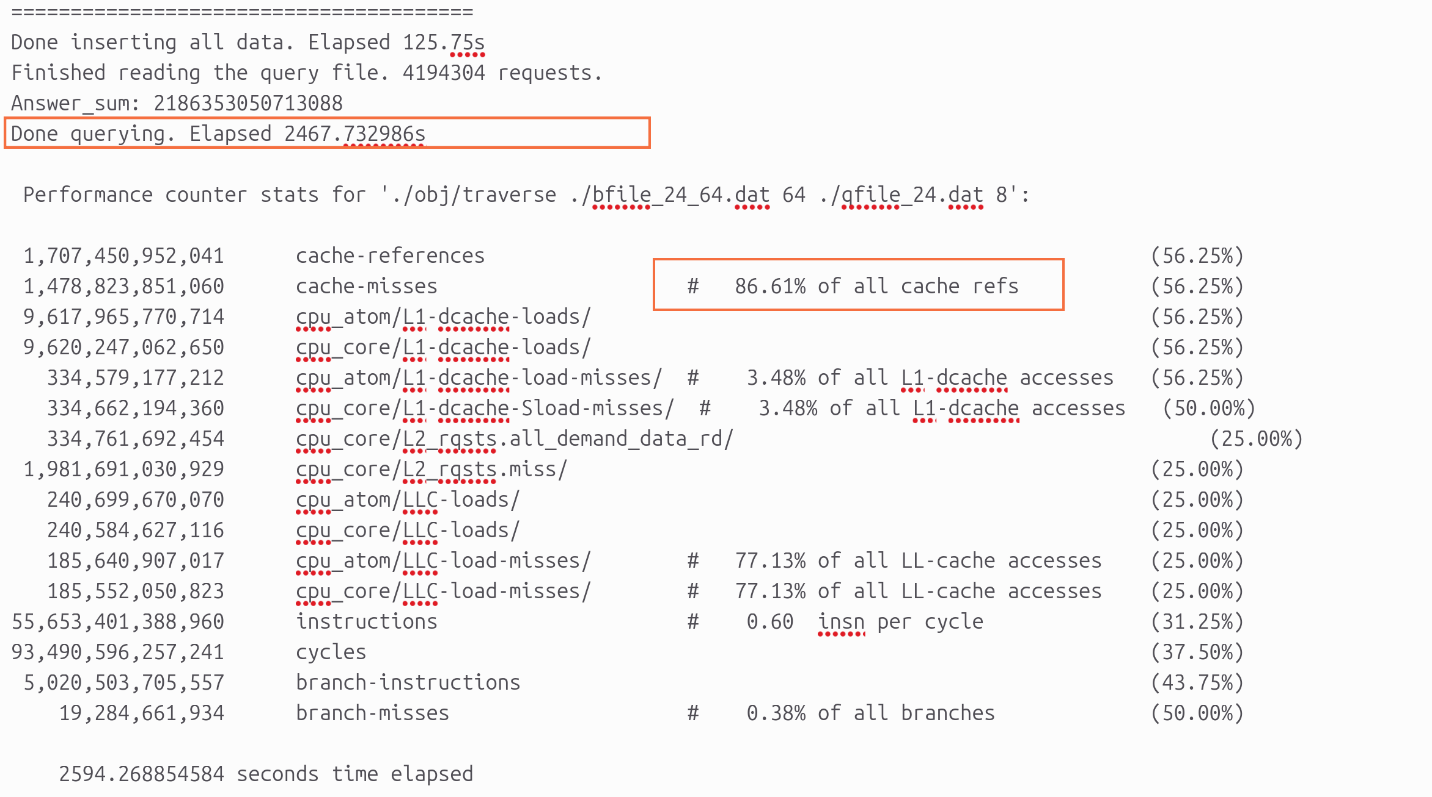
1. **Benchmarking**

For the benchmark tests of the Lab-1 code, I used the following hardware configuration:

* CPU: intel i9-13980HX with L1-cache 80 KB and L2-cache 2MB
* RAM: 32GB \*2 DDR5 at 5200 MHz
* Ubuntu 24.04

The original implementation, which uses a **Binary Search Tree (BST)**, runs for approximately **2600 seconds** on my laptop when processing a dataset of **10^24** records (refer to *"lab1-vector-src-3-result"*).

Performance profiling with perf revealed that the BST structure has a **high cache miss rate (around 87%)**, which significantly impacts query efficiency. The excessive cache misses result in prolonged execution times due to frequent data fetching from main memory.



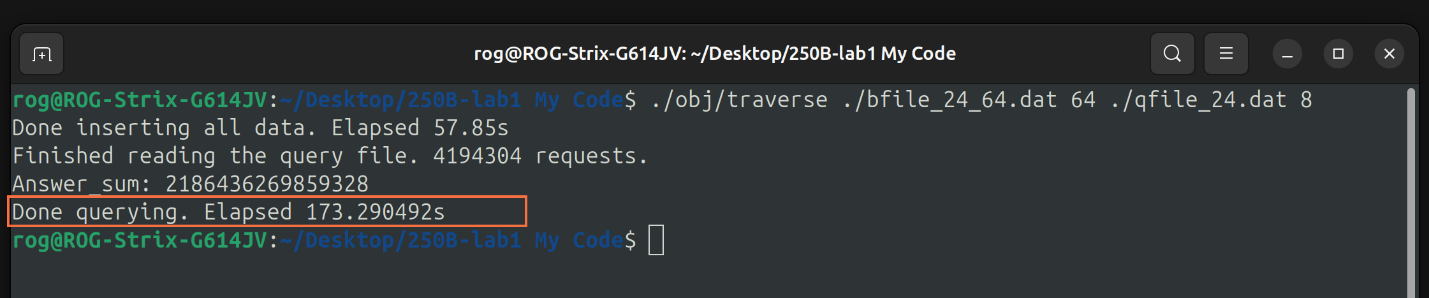
lab1-vector-src-3-result

1. **Key Improvements**

To optimize performance, I introduced several enhancements:

1. **Replacing BST with B+ Tree**  
   The BST was replaced with a B+ tree, which significantly improves insertion and search efficiency. The B+ tree offers a more cache-friendly structure, reducing tree traversal overhead.
2. **Optimized Manhattan Distance Calculation with AVX2**  
   The Manhattan Distance computation was optimized using the AVX2 instruction set, implementing SIMD parallelism to accelerate computations. This change greatly reduced the overall query execution time.
3. **Query Sorting and Thread Assignment Optimization**
   * Queries were sorted and then divided into continuous segments, ensuring that each thread processes queries in a contiguous key range.
   * This approach enhances cache efficiency by allowing better utilization of \_mm\_prefetch, as threads work on consecutive memory regions instead of randomly scattered queries.
4. **B+ Tree Node Order Tuning**
   * The **BPT\_ORDER** parameter in **tree.h** (which defines the number of children per node in the B+ tree) was tested with multiple values.
   * In the final implementation, **BPT\_ORDER = 256** was chosen, which slightly reduces query time at the cost of increased insertion time.

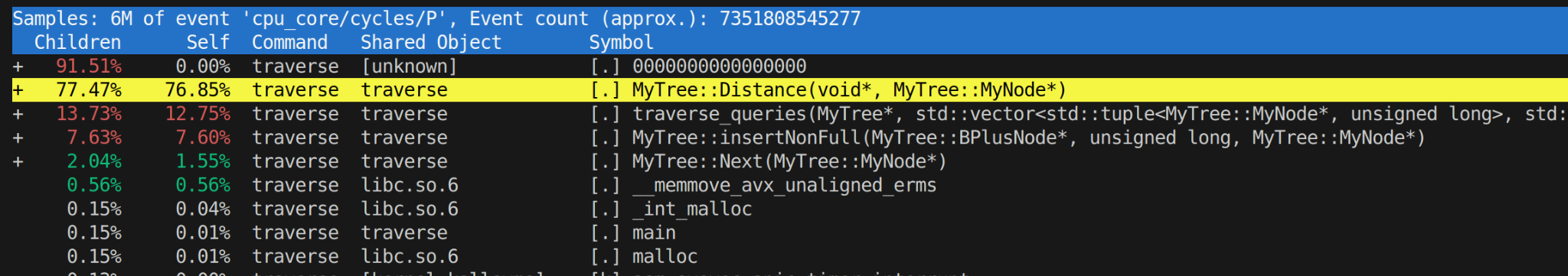
As a result of these optimizations, the final execution time **decreased to approximately 180 seconds**, achieving a **13× speedup** compared to the original BST-based implementation.



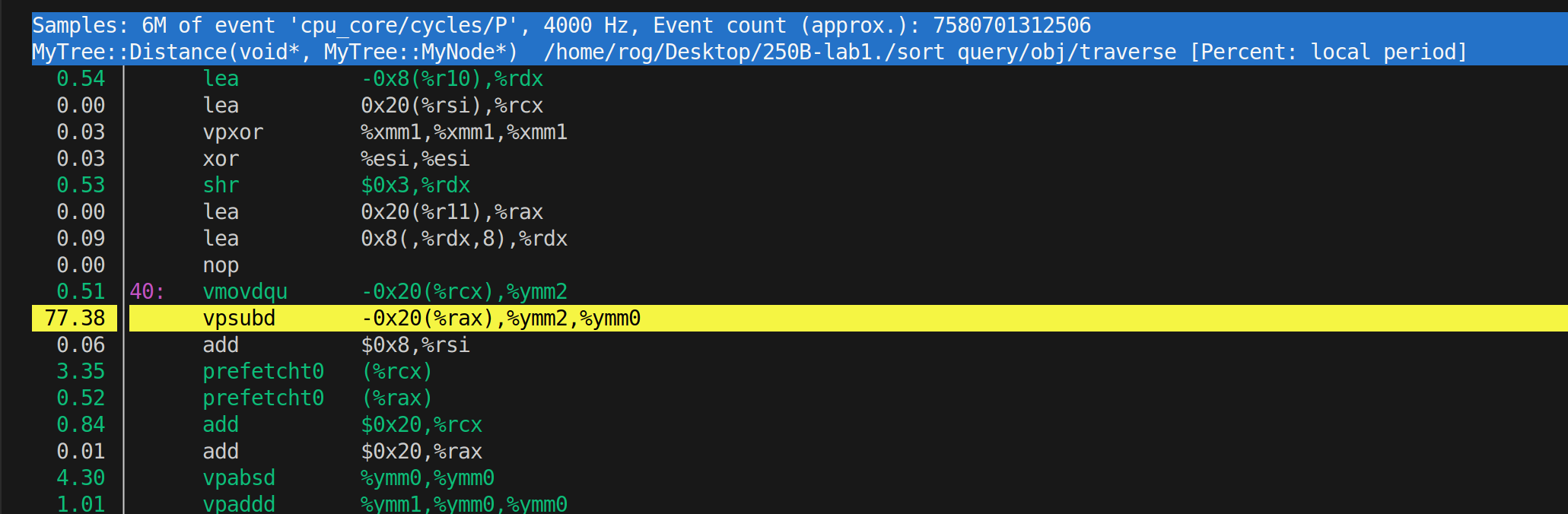
**Performance Analysis and Remaining Bottlenecks**

Despite the improvements, profiling results from perf highlight that the Distance function still dominates execution time:

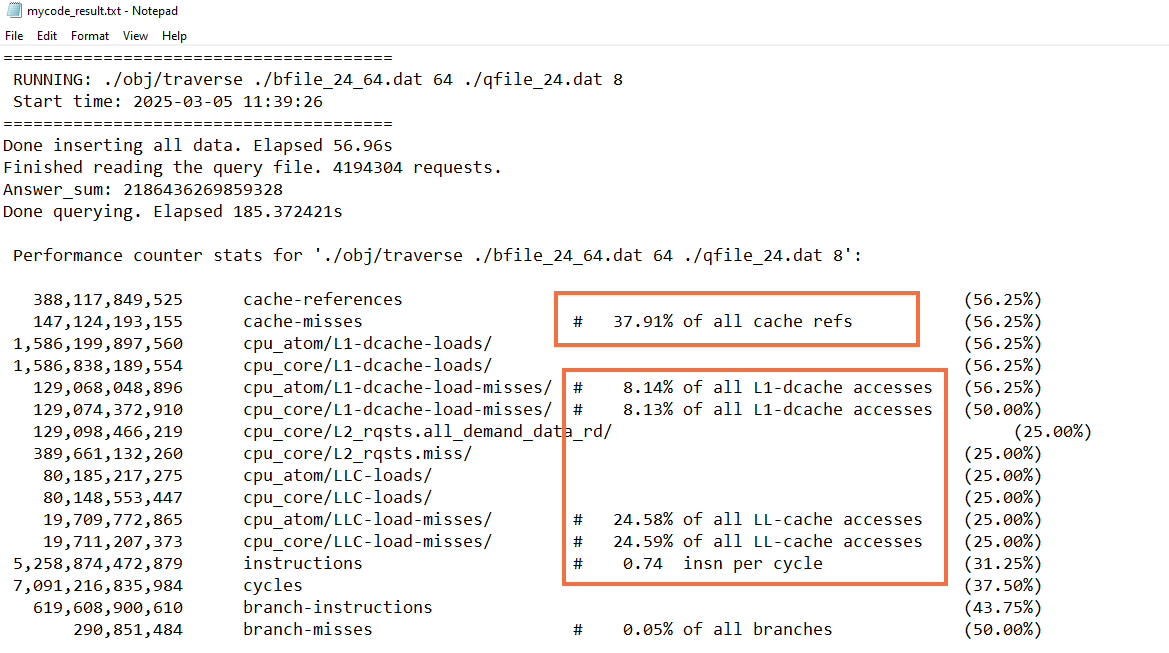
* **MyTree::Distance function:** ~80% of total CPU time
* **traverse\_queries function:** ~13% of total CPU time
* **insertNonFull function:** (only relevant during tree construction, not affecting queries)



A deeper analysis of the **Distance function** using instruction-level profiling reveals that the **"vpsubd" (vectorized subtraction) instruction alone consumes up to 77% of CPU time**. This suggests that the function is still **not fully optimized for cache usage**.



Moreover, the latest **cache performance records** (*"mycode\_result.txt"*) indicate that the total cache miss rate, while improved, remains **at 38%**. This suggests further room for **optimization in how data is loaded into the cache**.



mycode\_result.txt

1. **Conclusion**

The biggest challenge now is optimizing the **Distance function** to minimize data-loading overhead and better utilize cache memory. A key observation is that the computation is still **limited by data access latency rather than arithmetic operations**.

A promising direction for improvement was inspired by discussions with Yanjun Chen. While we did not share code, he proposed a **segment-based query approach**:

* Instead of executing each query independently, **partition the dataset into segments** after inserting it into the B+ tree.
* For each segment, identify the **set of queries that fall within that interval**.
* **Load the segment into cache once** and process all relevant queries before moving to the next segment.
* This ensures that **each data record is loaded exactly once**, avoiding redundant cache evictions and improving overall efficiency.

Implementing this approach would require **rewriting the query logic**, rethinking how results are accumulated, and reassigning thread workloads to handle **non-continuous** query execution patterns. While this demands significant debugging effort, it is an exciting avenue that could **drastically reduce cache misses and improve performance**.