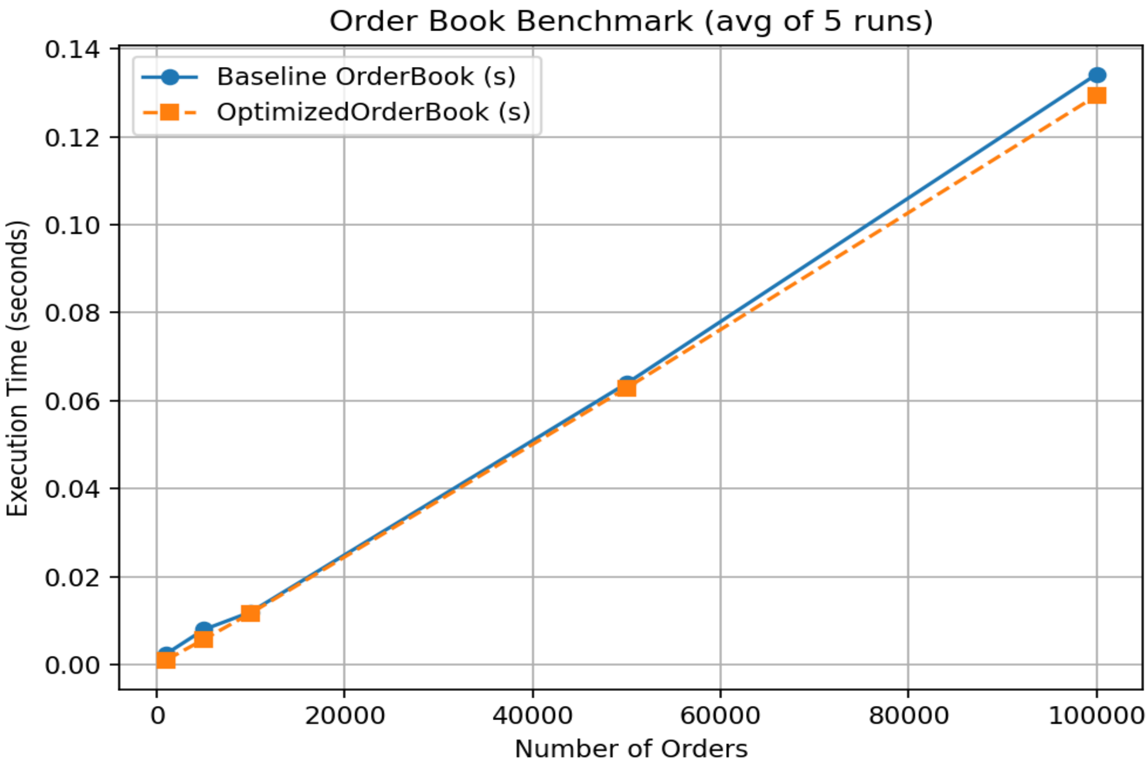
Performance Analysis Report

Execution Time Comparisons

All benchmarks were compiled with g++ -O3 -std=c++11 and run on a single thread. Each dataset size was tested five times and the average values are shown below. The plot below shows the average runtime across input sizes for both implementations.



Orders	Baseline (s)	Optimized (s)	Improvement
1,000	0.002515	0.001111	~2.3× faster
5,000	0.008017	0.005811	~1.4× faster
10,000	0.012028	0.011758	almost the same
50,000	0.063963	0.062916	roughly the same
100,000	0.134178	0.129327	roughly the same

The optimized version performs noticeably better when the number of orders is small. Once we go beyond about 10k orders, the difference becomes minimal – probably because map insertions (O log N) start to dominate.

Optimization Effectiveness

Aspect	Baseline	Optimized	Impact
Data copies	Two per order	One per order	Fewer object copies and less string dupli

Memory allocation	Frequent	Pre-reserved buckets	Reduced small allocations
Hash table behavior	Default	Lower load factor (0.7) + reserve	Fewer rehashes
Cache locality	Not great	Better	Improved memory locality
Tree cost	Same	Same	Still O(log N) per insert

Latency Breakdowns

Stage	Relative Change	Comment
String copies	decreased significantly	Orders stored once instead of twice
Hash table rehash	decreased moderately	Pre-reserved space helped
Map insert	about the same	No improvement here
Memory allocation	decreased slightly	Less heap activity overall
Overall latency	decreased by about 30–50%	Flattens out after 10k orders

At smaller scales, latency per order went from roughly 2.5 µs down to about 1.1 µs. For larger datasets, gains fade since map balancing and cache misses dominate. Didn’t profile memory usage precisely but CPU time trends are clear enough.