HFT Order Book Performance Analysis Report

FINM 32700

May 6, 2025

1 Introduction

This report details the performance analysis of two C++ order book implementations: an "Original OrderBook" using standard library containers directly and an "OptimizedOrderBook" incorporating advanced techniques such as memory pooling (for `Order` objects) and loop unrolling in benchmark execution. The objective was to evaluate the effectiveness of these optimizations in a simulated high-frequency trading (HFT) environment by measuring execution times for adding, modifying, and deleting orders across various volumes.

2 Methodology

- **Implementations:**
  - **Original OrderBook:** Utilized `std::map<double, std::unordered_map<std::string, Order>>` for price levels and `std::unordered_map<std::string, Order>` for direct ID lookup. Order objects were dynamically allocated as part of standard container operations.
  - **OptimizedOrderBook:** Utilized `std::map<double, std::unordered_map<std::string, Order*>>` and `std::unordered_map<std::string, Order*>`, storing pointers to `Order` objects. These `Order` objects were allocated from a custom `ObjectPool` to reduce dynamic memory allocation overhead. An atomic counter (`std::atomic<size_t>`) was used to track active orders, though its primary impact is in concurrent scenarios not explicitly benchmarked here for lock-freedom.
- **Benchmarking:**
  - A sequence of operations (Add, then Modify, then Delete) was performed for a given number of orders.
  - Execution times for each operation type (Add, Modify, Delete) and the total time for the sequence were measured using `std::chrono::high_resolution_clock`.
  - Order volumes tested: 1 000, 5 000, 10 000, 50 000, and 100 000.
  - The benchmark loops for the "OptimizedOrderBook" operations were unrolled by a factor of 4.
- **Environment:** Benchmarks were run on a single thread.

3 Execution Time Comparisons & Optimization Effectiveness

The following table summarizes the total execution time (in milliseconds) for the sequence of Add, Modify, and Delete operations for both implementations across different order volumes:

Key Observations:

- **Overall Improvement:** The `OptimizedOrderBook` consistently outperforms the `OriginalOrderBook` in terms of total execution time for the tested workload sequence, with improvements ranging from approximately 6% to 12%.

# Orders	Original Total Time (ms)	Optimized Total Time (ms)	% Improvement (Total)
1000.00	2.15	1.88	12.42
5000.00	7.12	6.48	8.93
10 000.00	10.71	9.53	10.94
50 000.00	41.38	38.81	6.21
100 000.00	84.68	76.49	9.67

Table 1: Total execution times for both order book implementations.

- **Scalability:** Both implementations show an increase in execution time as the number of orders increases, which is expected. The optimizations seem to provide a relatively consistent percentage improvement across the tested scales, rather than a dramatically different scaling curve. This suggests the core algorithmic complexity of map operations remains dominant, but constant factor overheads are reduced.
- **Optimization Effectiveness:**
  - *Memory Pooling:* The primary contributor to the observed speedup is likely the `ObjectPool` for `Order` objects. By pre-allocating memory and managing `Order` instances within the pool, the overhead associated with frequent dynamic memory allocations (`new`/`delete`) by standard containers is significantly reduced. This also improves data locality, potentially leading to better cache performance.
  - *Loop Unrolling:* While applied to the benchmark loops themselves (not internal order book logic), loop unrolling can reduce loop control overhead. Its direct impact on the order book's internal performance is indirect but contributes to faster benchmarking execution.
  - *Atomic Counter:* In this single-threaded benchmark, the `std::atomic` counter for active orders adds negligible overhead and offers no direct performance benefit over a simple integer. Its value lies in thread-safe counting for potential concurrent extensions.

4 Latency Breakdowns (Per Operation Type)

4.1 Add Operation

# Orders	Original Add (ms)	Optimized Add (ms)	Difference (Orig - Opt)
1000.000	0.495	0.523	−0.028
5000.000	2.079	2.300	−0.221
10 000.000	3.042	3.209	−0.167
50 000.000	11.777	12.692	−0.915
100 000.000	22.828	26.351	−3.523

Table 2: Add operation latency comparison.

Observation: Surprisingly, the `OptimizedOrderBook`'s *addOrder* operation is consistently slower than the `OriginalOrderBook`'s.

Potential Reasons:

1. *Object Pool Allocation Overhead:* While the pool aims to be faster than global allocations, its internal management (e.g., list operations) may introduce overhead.
2. *Pointer Indirection:* Storing pointers to `Order` adds an extra level of indirection.

3. *Existence Check:* The optimized version checks `orderLookup.count(id)` before allocation, adding a lookup.

4. *Measurement Noise:* Small absolute differences can be susceptible to system noise.

4.2 Modify Operation

# Orders	Original Modify (ms)	Optimized Modify (ms)	Difference (Orig - Opt)	% Improvement
1000.000	0.963	0.627	0.336	34.890
5000.000	2.700	1.910	0.790	29.260
10 000.000	4.118	2.918	1.200	29.140
50 000.000	16.138	13.189	2.949	18.270
100 000.000	33.796	23.571	10.225	30.260

Table 3: Modify operation latency comparison.

Observation: The `OptimizedOrderBook` shows significant improvement in *modifyOrder*, consistently around 18%–35%.

Reasoning: Modifications in the original version involve deallocation and reallocation, whereas the optimized version updates in place or re-links pooled objects, reducing memory churn.

4.3 Delete Operation

# Orders	Original Delete (ms)	Optimized Delete (ms)	Difference (Orig - Opt)
1000.000	0.691	0.732	−0.041
5000.000	2.339	2.272	0.067
10 000.000	3.544	3.407	0.137
50 000.000	13.469	12.932	0.537
100 000.000	28.053	26.565	1.488

Table 4: Delete operation latency comparison.

Observation: The `deleteOrder` performance is mixed, with slight improvements at larger scales.

Reasoning: Both versions perform map erasure; the optimized version uses pool deallocation, showing benefits when many objects return to the pool.

5 Analysis of Bottlenecks

- **Memory Allocation/Deallocation:** The significant modify improvement confirms memory churn as a bottleneck, effectively mitigated by pooling. The slower add suggests pool strategy refinement is needed.
- **Map Operations:** Underlying map and unordered_map operations remain the core workload; their complexity dictates scalability.
- **String Operations:** Hashing and comparisons for string IDs contribute to latency and are not directly addressed by current optimizations.