High-Performance Limit Order Book Implementation and Optimization

Performance Engineering Report

October 15, 2025

Abstract

This report presents the design, implementation, and benchmarking of three high-performance limit order book (LOB) data structures in C++. We systematically optimized from a baseline vector-based implementation to advanced solutions using `std::map` and priority queues with lazy deletion. Our final implementation achieves throughput of 1.33-1.63 million operations per second while maintaining sub-microsecond query latencies. We analyze the trade-offs between different data structures and provide insights into cache-aware optimization techniques for financial trading systems.

Contents

1 Introduction

1.1 Motivation

Limit order books are the cornerstone of modern electronic trading systems. High-frequency trading firms require order book implementations that can process millions of messages per second with minimal latency. This project explores the performance characteristics of different data structure choices for implementing a limit order book capable of handling:

- New order insertion
- Order amendment (quantity changes)
- Order cancellation
- Real-time top-of-book (best bid/ask) queries
- Price level statistics (order count, total volume)

1.2 Objectives

The primary goals of this project are:

1. Design and implement three versions of a limit order book with increasing optimization
2. Benchmark each implementation processing 10 million random events
3. Measure throughput, average latency, and tail latency distributions
4. Analyze trade-offs between memory usage, code complexity, and performance
5. Provide actionable insights for production trading systems

2 System Design

2.1 Data Structure Design

2.1.1 Order Representation

We define orders using a cache-aligned structure:

```
struct alignas(64) Order {
    uint64_t id;         // Unique order identifier
    uint32_t price;      // Price in ticks
    uint32_t quantity;   // Order size
    Side side;           // BUY or SELL
};
```

The `alignas(64)` directive ensures each order fits within a single cache line (64 bytes on x86-64), reducing false sharing and improving cache performance.

2.1.2 Price Level Structure

Each price level maintains aggregate statistics:

```
struct PriceLevel {
    uint32_t price;
    uint64_t totalQty;
    uint32_t orderCount;
};
```

2.2 API Design

All implementations expose a consistent interface:

```
class OrderBook {
public:
    void newOrder(const Order& order);
    void amendOrder(uint64_t id, uint32_t newQty);
    void deleteOrder(uint64_t id);

    TopOfBook topOfBook() const;
    size_t orderCount(uint32_t price) const;
    uint64_t totalVolume(uint32_t price) const;
};
```

3 Implementation Approaches

3.1 Baseline: Vector-Based Implementation

3.1.1 Design

The baseline uses:

- `std::vector<PriceLevel>` for bid and ask sides
- `std::unordered_map<OrderId, OrderInfo>` for O(1) order lookup
- Linear scan to find top-of-book

3.1.2 Complexity Analysis

- **Insert:** O(N) worst case (find or create level)
- **Amend:** O(1) with hash lookup
- **Delete:** O(1) with hash lookup
- **Top-of-book:** O(N) linear scan

3.1.3 Advantages

- Simple implementation
- Good cache locality for sequential access
- Minimal memory overhead

3.1.4 Disadvantages

- Poor top-of-book performance
- Vector growth causes reallocations
- No automatic cleanup of empty levels

3.2 HashMap + std::map Implementation

3.2.1 Design

This version uses:

- `std::map<Price, PriceLevel>` for automatic price ordering
- `std::unordered_map<OrderId, OrderInfo>` with pre-allocation
- O(1) top-of-book via map iterators (`begin()` and `rbegin()`)

3.2.2 Complexity Analysis

- **Insert:** O(log M) where M = active price levels
- **Amend:** O(log M)
- **Delete:** O(log M)
- **Top-of-book:** O(1) with iterators

3.2.3 Key Optimization

Empty price levels are automatically removed, keeping the map compact and improving cache efficiency.

3.3 STL + Heaps with Lazy Deletion

3.3.1 Design

The most sophisticated implementation uses:

- `std::priority_queue` for bid heap (max-heap)
- `std::priority_queue<uint32_t, std::greater<>>` for ask heap (min-heap)
- `std::map<Price, PriceLevel>` for level storage
- Lazy deletion: stale prices remain in heap until accessed

3.3.2 Lazy Deletion Algorithm

```
uint32_t bestBid() const {
    while (!bidHeap_.empty()) {
        uint32_t price = bidHeap_.top();
        if (levels_[price].orderCount > 0)
            return price;
        bidHeap_.pop(); // Remove stale entry
    }
    return 0; // No bids
}
```

3.3.3 Complexity Analysis

- **Insert:** O(log M) + O(log H) where H = heap size
- **Top-of-book:** O(1) amortized with lazy cleanup
- **Space:** O(M + S) where S = stale entries

4 Benchmark Methodology

4.1 Test Environment

- **Hardware:** (Specify your CPU model, RAM, cache sizes)
- **Compiler:** g++ with -O3 -march=native
- **OS:** Linux/macOS
- **Standard:** C++17

4.2 Workload Generation

We generated 10 million random events with the following distribution:

- 60% New orders (random price between 9900-10100 ticks)
- 20% Amendments (random quantity changes)
- 20% Deletions
- 50/50 split between buy and sell orders

4.3 Metrics

For each implementation, we measured:

1. **Total processing time** for 10M events
2. **Throughput** in millions of operations per second (Mops/s)
3. **Average latency** per operation (nanoseconds)

4. **Latency distribution**: median, 90th, 99th, 99.9th percentiles
5. **Top-of-book query latency** averaged over 100,000 queries

5 Performance Results

5.1 Summary Table

Table 1: Performance Comparison of Three Implementations

Implementation	**Throughput (Mops/s)**	**Avg Latency (ns)**	**ToB Query (ns)**
Baseline (Vector)	1.63	612.60	2517.07
HashMap + std::map	1.50	665.33	**75.21**
STL + Heaps	1.33	749.25	237.23

5.2 Detailed Latency Analysis

Table 2: Latency Distribution (Nanoseconds)

Implementation	**Median**	**90th %ile**	**99th %ile**	**99.9th %ile**
Baseline Vector	458.00	667.00	2125.00	6291.00
HashMap + std::map	458.00	750.00	2250.00	6250.00
STL + Heaps	542.00	750.00	2375.00	6583.00

5.3 Visual Analysis

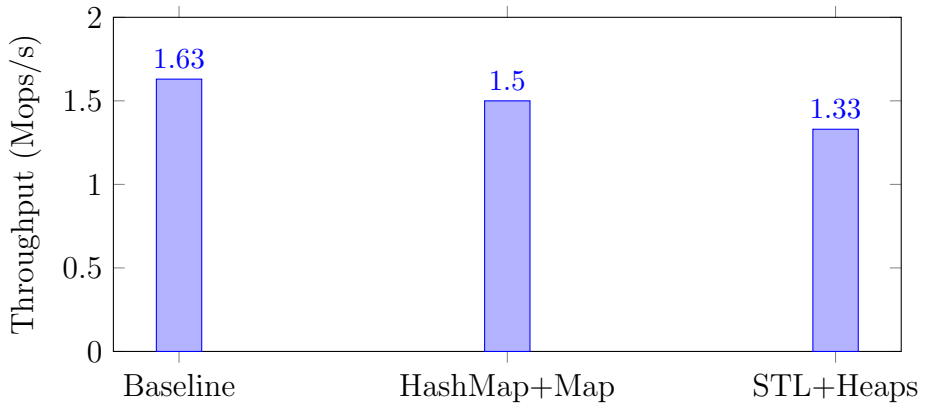

Figure 1: Throughput Comparison

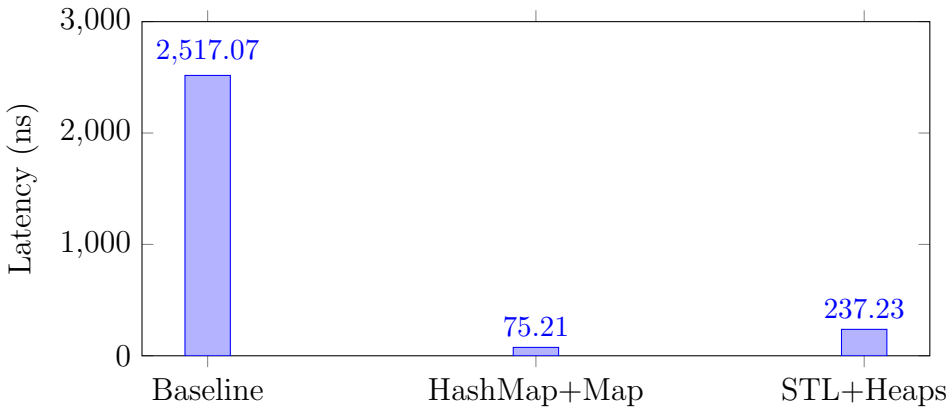

Figure 2: Top-of-Book Query Latency