

A field guide to production trading
systems

DETERMINISTIC MATCHING ENGINES IN C++23/26

**Lock-Free Order Books, Cache-Line Aligned
Price-Time Priority, and Zero-Jitter Trade
Execution.**

Weston Ashgrove

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

This book is intended for informational and educational purposes only. While every effort has been made to ensure the accuracy and completeness of the information contained herein, the author and publisher make no representations or warranties with respect to the accuracy or completeness of the contents and specifically disclaim any implied warranties.

The design patterns, benchmarks, and code examples described in this book are provided for educational purposes and should be independently tested and validated before use in any production trading environment. Neither the author nor the publisher shall be liable for any financial loss, trading loss, or other damages arising from the use of the information contained in this book.

All trademarks, service marks, and trade names referenced in this book are the property of their respective owners and are used for identification purposes only. Use of these names does not imply endorsement.

First Edition: 2026

What You'll Build

By the end of this book, you will have built a complete, production-grade matching engine capable of processing over one million orders per second with sub-100ns p99.9 tail latency. Starting from correct, provably fair matching logic, you will progressively engineer out every source of non-determinism: lock contention, false sharing, heap allocation, and OS-level jitter, until the engine's behavior is fully deterministic and its performance is fully measured, not assumed. The final chapters prove the engine works under real conditions: deterministic replay, property-based testing, and fault recovery without losing order integrity. You will finish with a working reference implementation, a full benchmarking harness, and a clear understanding of exactly where every nanosecond in the system goes.

Prerequisites

You should be comfortable with C++ at a working level, including templates, RAII, and move semantics. This book does not cover basic C++ syntax. A Linux environment is assumed throughout, and tools such as `perf`, `taskset`, and `numactl` are used extensively; prior familiarity is helpful but not required, as each is introduced in context the first time it's used. No prior experience with lock-free programming or low-latency systems is assumed.

How to Use This Book: The Latency Budget Ledger

Every chapter in this book closes with a Latency Budget Ledger, a short running record of the nanoseconds identified and removed from the hot path at that stage, along with the technique responsible.

This exists because low-latency engineering is rarely one dramatic optimization. It is the accumulation of many small, measured decisions: a struct realigned to avoid false sharing, an allocation moved out of the hot path, a lock replaced with a compare-and-swap loop. By the end of the book, the ledger becomes a complete record of the engine's performance journey, from a naive first implementation to a system with sub-100ns tail latency, and a template you can apply to your own systems.

Each entry follows the same short format:

- **Where:** the component or code path being optimized
- **Cost:** the latency identified and its cause
- **Fix:** the specific technique used to remove it

By Chapter 11, the ledger doubles as a complete latency-optimization checklist for future low-latency C++ systems.

Table of Contents

Chapter 1	10
The Anatomy of a Production Failure	10
1.1 A real matching engine outage, reconstructed minute by minute	10
1.2 Why naive implementations pass every demo and fail in week two	12
1.3 Defining "deterministic" precisely: guarantees versus non-guarantees	13
1.4 Setting up the benchmark environment (perf, taskset, numactl, compiler flags)	15
Chapter 2	18
Order Book Data Structures Under Real Volume	18
2.1 Array-based book: cache-friendly but $O(n)$ insertion	18
2.2 Skip-list book: $O(\log n)$ with pointer-chasing cost	20
2.3 Red-black tree book: balanced but poor cache locality	22
2.4 Price-level aggregation versus per-order tracking trade-offs	24
2.5 Benchmark: 1M orders per second across all three structures, raw numbers	26
Chapter 3	Error! Bookmark not defined.
Order Types and Matching Semantics	Error! Bookmark not defined.
3.1 Limit and market orders: the baseline matching logic	Error! Bookmark not defined.
3.2 Stop and stop-limit orders: trigger mechanics and edge cases	Error! Bookmark not defined.
3.3 Iceberg orders: hidden quantity handling without leaking book state	Error! Bookmark not defined.
3.4 Post-only orders: rejecting orders that would take liquidity	Error! Bookmark not defined.
3.5 Order amendment and cancel/replace semantics	Error! Bookmark not defined.
Chapter 4	Error! Bookmark not defined.
Price-Time Priority and Fairness Guarantees	Error! Bookmark not defined.
4.1 FIFO matching semantics and where implementations quietly break them	Error! Bookmark not defined.
4.2 Partial fills and priority preservation	Error! Bookmark not defined.

4.3 Amendments that should, and shouldn't, reset time priority.....	Error!
Bookmark not defined.	
4.4 War story: the fairness bug that triggered a regulatory inquiry.....	Error!
Bookmark not defined.	
Chapter 5.....	Error! Bookmark not defined.
Lock-Free Design with Compare-and-Swap.....	Error! Bookmark not defined.
5.1 Where mutexes silently introduce priority inversion..	Error! Bookmark not defined.
5.2 std::atomic and CAS loops: full implementation walkthrough.....	Error!
Bookmark not defined.	
5.3 The ABA problem and hazard pointers as a fix .	Error! Bookmark not defined.
5.4 Memory ordering: relaxed, acquire/release, and seq_cst, and when each is correct.....	Error! Bookmark not defined.
Chapter 6.....	Error! Bookmark not defined.
Cache-Line Alignment and False Sharing ..	Error! Bookmark not defined.
6.1 Struct layout for order and price-level data	Error! Bookmark not defined.
6.2 alignas(64) and padding strategy.....	Error! Bookmark not defined.
6.3 Measuring false sharing with perf c2c .	Error! Bookmark not defined.
6.4 Before and after comparison: 340ns reduced to 40ns on a real workload.....	Error! Bookmark not defined.
Chapter 7.....	Error! Bookmark not defined.
Memory Pools and Zero-Allocation Hot Paths.....	Error! Bookmark not defined.
7.1 Why malloc and new in the hot path are a silent latency bomb....	Error!
Bookmark not defined.	
7.2 Pre-allocated pool design for orders and price levels	Error! Bookmark not defined.
7.3 Custom allocator implementation: std::pmr versus hand-rolled ...	Error!
Bookmark not defined.	
7.4 Verifying zero allocation with allocator instrumentation	Error!
Bookmark not defined.	
Chapter 8.....	Error! Bookmark not defined.

Eliminating OS and Runtime Jitter.....	Error! Bookmark not defined.
8.1 CPU pinning with taskset and sched_setaffinity	Error! Bookmark not defined.
8.2 SCHED_FIFO real-time scheduling and priority inversion risks.	Error! Bookmark not defined.
8.3 NUMA-aware memory placement with numactl	Error! Bookmark not defined.
8.4 Interrupt isolation using isolcpus and irqaffinity	Error! Bookmark not defined.
8.5 The nine sources of jitter most engineers never check.....	Error! Bookmark not defined.
Chapter 9.....	Error! Bookmark not defined.
Measuring Tail Latency, Not Averages	Error! Bookmark not defined.
9.1 Why p50 lies and p99.9/p99.99 tell the truth.....	Error! Bookmark not defined.
9.2 Building a load generator that simulates realistic burst patterns ...	Error! Bookmark not defined.
9.3 Reading and interpreting latency histograms using HDR histogram	Error! Bookmark not defined.
9.4 Correlating tail spikes to specific code paths.....	Error! Bookmark not defined.
Chapter 10	Error! Bookmark not defined.
Deterministic Replay and Property-Based Testing .	Error! Bookmark not defined.
10.1 Event sourcing the order flow for exact replay	Error! Bookmark not defined.
10.2 Fuzzing the matching logic with libFuzzer for edge-case discovery	Error! Bookmark not defined.
10.3 Proving byte-identical output across 10,000 replayed runs.....	Error! Bookmark not defined.
10.4 Property-based tests for invariants such as price-time priority and quantity conservation	Error! Bookmark not defined.
Chapter 11	Error! Bookmark not defined.
Fault Recovery Without Losing Determinism	Error! Bookmark not defined.

11.1 State reconstruction from an event log after a crash	Error! Bookmark not defined.
11.2 Resuming mid-session without breaking price-time priority	Error! Bookmark not defined.
11.3 Chaos-testing the recovery path using tc netem and fault injection	Error! Bookmark not defined.
11.4 Recovery time objective benchmarking	Error! Bookmark not defined.
Appendix A	Error! Bookmark not defined.
Full annotated reference implementation ...	Error! Bookmark not defined.
A.1 Core types and cache-aligned layout (2.1, 2.4, 6.1, 6.2).....	Error! Bookmark not defined.
A.2 Object pool with hazard-pointer-safe reclamation (5.3, 7.2)	Error! Bookmark not defined.
A.3 Lock-free resting-order queue (5.2, 5.3, 5.4)	Error! Bookmark not defined.
A.4 Price level (2.1, 6.1) and the array book (2.1, 2.5).....	Error! Bookmark not defined.
A.5 Matching: limit, market, and the shared fill loop (3.1)	Error! Bookmark not defined.
A.6 Order-type entry points (3.2, 3.3, 3.4, 3.5).....	Error! Bookmark not defined.
A.7 Event logging hook (10.1) wired into every entry point.....	Error! Bookmark not defined.
A.8 Deployment scaffolding referenced, not reproduced	Error! Bookmark not defined.
Appendix B	Error! Bookmark not defined.
Cross-architecture benchmark data (x86-64, ARM/Graviton)	Error! Bookmark not defined.
B.1 Why this needs its own appendix rather than a footnote	Error! Bookmark not defined.
B.2 Memory ordering: TSO versus weakly-ordered .	Error! Bookmark not defined.
B.3 CAS primitive: cmpxchg versus LL/SC and LSE atomics	Error! Bookmark not defined.

B.4 Cache line size and false-sharing implications	Error! Bookmark not defined.
B.5 NUMA and interconnect topology	Error! Bookmark not defined.
B.6 Toolchain equivalents for Chapter 6 and Chapter 8's verification	Error! Bookmark not defined.
B.7 Net guidance.....	Error! Bookmark not defined.
Appendix C	Error! Bookmark not defined.
Five real production post-mortems, anonymized	Error! Bookmark not defined.
C.1 The mutex that only failed on Tuesdays	Error! Bookmark not defined.
C.2 The false-sharing regression that shipped in a refactor.....	Error! Bookmark not defined.
C.3 The allocator that worked fine until the pool filled...	Error! Bookmark not defined.
C.4 The NUMA mismatch nobody could reproduce in staging.....	Error! Bookmark not defined.
C.5 The replay that proved the incident wasn't what it first looked like	Error! Bookmark not defined.
Appendix D.....	Error! Bookmark not defined.
Quick reference of latency costs for common operations	Error! Bookmark not defined.
D.1 Memory hierarchy	Error! Bookmark not defined.
D.2 Atomics and synchronization.....	Error! Bookmark not defined.
D.3 Scheduling and OS.....	Error! Bookmark not defined.
D.4 Branch and instruction-level costs.....	Error! Bookmark not defined.
D.5 Composite / end-to-end figures from the book's own benchmarks	Error! Bookmark not defined.
D.6 Using this table.....	Error! Bookmark not defined.

PART I

The Correctness Baseline

*Before optimizing anything, the engine must be
provably correct.*

Chapter 1

1.1 The Anatomy of a Production Failure

1.1 A real matching engine outage, reconstructed minute by minute

(The incident below is a composite, assembled from failure patterns that recur across postmortems in this space. Treat the timestamps and numbers as representative of a typical failure mode, not as a report of one specific engine.)

The engine had been in production for four months. It had survived two earnings seasons and a rate announcement without incident. The order book was a single `std::map<Price, PriceLevel>` protected by one `std::mutex`, with each `PriceLevel` holding a `std::deque<Order>` for time priority within that level. Every insert, cancel, and match acquired the same lock. In staging, under a synthetic load generator firing a steady 8,000 orders per second, median match latency sat at 4 microseconds and nobody looked closer than the median.

08:58 — Pre-open order flow builds normally. The book absorbs roughly 3,000 messages per second, well inside its tested range. `perf top` on the engine host, if anyone had been watching it, would have shown the matching thread spending under 2% of its time blocked on the mutex.

09:30:00 — The exchange's opening auction clears and continuous trading begins. Order flow jumps to 41,000 messages per second within the first eight seconds, driven by a cluster of algorithmic participants reacting to the auction price. This is roughly five times the load the engine had ever been tested against.

09:30:04 — Lock contention on the book mutex starts costing real time. Under low contention, an uncontended `std::mutex::lock()` costs on the order of 20 nanoseconds. Under contention, the same call routes through the kernel futex path, and each acquisition now costs somewhere in the range of 1 to 5

microseconds depending on how many threads are queued behind it. With four matching-adjacent threads (the matcher itself, a risk-check thread, a market-data publisher, and a logger) all touching the same lock, the queue depth behind the mutex starts to grow rather than drain.

09:30:11 — The Linux scheduler preempts the thread holding the lock, mid-match, in favor of a housekeeping thread with equal priority. Nothing in the system had pinned threads to cores or granted the matcher a real-time scheduling class, so the kernel treats it like any other thread when its time slice runs out. Every other thread waiting on that mutex now stalls for however long the scheduler leaves the housekeeping thread running, typically several hundred microseconds, sometimes low milliseconds under load. This is priority inversion in its plainest form: a thread with the least urgent job on the system is, for a moment, the only thing allowed to run.

09:30:19 — The inbound order queue, which the network thread fills independently of whether the matcher is keeping up, passes 30,000 unprocessed messages. The matcher is still consistent internally, every message it does process is matched correctly, but it is now roughly 700 milliseconds behind the exchange's view of the book. Market data published downstream reflects the engine's stale internal state, not current market reality.

09:30:27 — A market-making client's cancel-replace for a resting order arrives and queues behind the backlog. The original order, priced off a market that has since moved eleven cents, gets matched before the cancel is processed. The fill is correct by the letter of price-time priority within the engine's own (stale) view of the book. It is a materially bad fill by the standard of what the market actually looked like at the moment the client sent the cancel.

09:31:40 — On-call is paged, not by a crash or an error log, since none exist, but by a client calling to ask why their fill prices don't match what any other venue was showing in the same window.

09:34 — Engineers pull perf record against the live process and confirm heavy time in `futex_wait`. The book itself has not corrupted. No invariant has broken. The engine has simply been running an average of 600 to 900 milliseconds behind real time for over four minutes, quietly, without any metric crossing a threshold that anyone had configured an alert on.

09:38 — The engine is failed over to a hot standby and the backlog is drained offline. Total elapsed time from first contention to failover: eight minutes. Total bad fills, once reconciled: several hundred, all individually plausible, none individually alarming, all wrong in aggregate.

Nothing about this failure shows up in a stack trace. The book was thread-safe in the narrow sense that no two threads ever corrupted its memory. It was not correct in the sense that mattered, which was staying close enough to real time that price-time priority meant what clients assumed it meant. That gap, between "does not crash" and "behaves correctly under load," is the entire subject of this

book, and it is where naive matching engine implementations go to die in week two rather than in the demo.

1.2 Why naive implementations pass every demo and fail in week two

Every naive matching engine passes its demo for the same reason: a demo has no contention. One order arrives, gets matched or rested, and the next one doesn't show up until the first has finished. Under that condition, a `std::mutex` around a `std::map` is indistinguishable from a lock-free ring buffer, because nothing is ever waiting. The architecture gets validated against a load profile that will never recur once the engine touches real order flow, and the team walks away believing the design is sound.

The failure mode in 1.1 didn't come from a bug in the matching logic. The matching logic was correct on every single message. It came from an assumption baked into the data structure choice that nobody stated out loud: that lock acquisition is free, or close enough to free that it doesn't need to be measured. That assumption holds at 3,000 messages per second on four threads. It stops holding somewhere between 15,000 and 25,000 messages per second, and where exactly it stops holding depends on thread count, core topology, and how much work happens inside the critical section, which is precisely why it never shows up until real flow finds the threshold nobody tested.

Three patterns account for most of the gap between demo and week two.

The first is testing throughput instead of testing contention. A load generator firing orders at a fixed rate from a single thread tells you almost nothing about how the engine behaves when four or five threads are genuinely racing for the same lock. Real production traffic is bursty by nature, an auction clears, a headline hits, a hedging algorithm fires, and the interesting failures live in the burst, not the average. A benchmark that reports mean latency at steady load will look identical for a design that degrades gracefully under contention and one that falls off a cliff at a specific concurrency threshold. You have to construct the burst deliberately, and you have to measure what happens to p99.9 and p99.99, not p50, because the median is the one number in the system that lies most convincingly.

The second is confusing "no crash" with "correct." The engine in 1.1 never corrupted its book. Every `std::map` operation completed. Every deque push and pop was internally consistent. Thread safety in the narrow sense, no data race, no torn write, was fully satisfied, and it satisfied exactly nothing about whether the book's view of the world matched the exchange's view of the world in real time. Determinism is a stronger property than thread safety, and it's the one

that actually matters for price-time priority. A book can be perfectly thread-safe and still be running most of a second behind reality, and nothing about a thread-safety audit will catch that.

The third is not budgeting for scheduler behavior at all. Most engineers building their first matching engine think about locks, and maybe about lock-free alternatives, and stop there. They don't think about what the OS scheduler does to a thread holding a lock when its time slice expires, because on a laptop, with two threads and no real load, it never matters. Under real order flow with real thread counts, an unpinned thread with no real-time scheduling class gets preempted at the worst possible moment often enough that priority inversion isn't an edge case, it's a recurring cost you're paying every single trading session without knowing it. Chapter 5 deals with the lock-free design that avoids most of this, and Chapter 8 deals with the scheduler and CPU affinity work that closes the rest of the gap. Neither one matters if you don't first understand why the demo can't see the problem.

The pattern underneath all three is the same: a demo exercises the code path, not the failure mode. The code path was never wrong. The failure mode lives entirely in what happens when multiple things want the same resource at the same instant, and a design that has never been asked that question has no evidence of correctness at all, regardless of how many times it's been run.

1.3 Defining "deterministic" precisely: guarantees versus non-guarantees

"Deterministic" gets used loosely enough in this field that it's worth pinning down before it appears in another 300 pages of chapter titles. It does not mean the engine runs in constant time. It does not mean every code path executes in the same number of nanoseconds regardless of input. It means something narrower and more useful: given the same sequence of inbound events, the engine produces the same sequence of matches, fills, and book states, every time, on every run. Determinism is a property of outcomes, not of timing. That distinction matters because the two get conflated constantly, and the conflation leads teams to solve the wrong problem. An engine can have highly variable per-message latency, some matches taking 80 nanoseconds and others taking 400 under cache pressure, and still be fully deterministic, because variable timing didn't change which order matched which, at what price, or in what sequence. Conversely, an engine can have beautifully flat, predictable latency and still be non-deterministic, if the order in which concurrent threads observe and apply events isn't fixed by the design. The engine in 1.1 wasn't slow because it was non-deterministic. It became effectively non-deterministic, from the client's perspective, because its notion of "the current book" drifted far enough

from real time that price-time priority stopped meaning what price-time priority is supposed to mean. The guarantee that broke wasn't a timing guarantee. It was an ordering guarantee.

What determinism actually guarantees, precisely:

Given a fixed, ordered sequence of inbound events (new orders, cancels, amends, and the wall-clock or sequence-number boundaries between them), the resulting sequence of matches is fully reproducible. Replay the same event log twice and you get byte-identical trade output both times. This is the guarantee Chapter 10 builds tooling around, and it's the one that makes postmortems possible at all, since without it you can't distinguish a real bug from a one-off scheduling artifact.

Given that same fixed sequence, price-time priority is preserved exactly: at a given price level, the order that arrived first fills first, full stop, with no scenario in which two orders at the same price level fill out of arrival sequence because of which CPU core happened to process them.

Partial fills and quantity conservation hold as invariants regardless of how many threads touch the book concurrently: total quantity in equals total quantity matched plus total quantity resting, always, with no dropped or duplicated units under any interleaving of concurrent operations.

What determinism does not guarantee, and this list matters as much as the first one:

It does not guarantee that any two runs take the same amount of wall-clock time. Two replays of an identical event log can differ by microseconds or more in total execution time depending on cache state, core frequency scaling, and what else is running on the host. If your test suite asserts on absolute timing rather than on match sequence and fill correctness, the test suite is wrong, not the engine.

It does not guarantee fairness across price levels that weren't contended in the first place. If two orders never actually compete for the same fill, determinism says nothing about which one a human might have expected to go first based on some other criterion. Determinism is about reproducing a specific, well-defined ordering rule consistently, not about producing outcomes a person finds intuitively fair.

It does not guarantee low latency. A fully deterministic engine can still be slow. Determinism and performance are orthogonal properties that happen to share an enemy: uncontrolled concurrency tends to threaten both at once, which is why they get treated together in this book, but fixing one doesn't automatically fix the other. Part II covers the performance side specifically because a correct, deterministic engine that takes 50 microseconds per match is a different kind of unacceptable than a fast one that occasionally reorders fills, and they get diagnosed with different tools.

Every technique in the rest of this book earns its place by which side of this line it addresses. Lock-free CAS loops, cache-line alignment, memory pools, none

of that changes what determinism means. It changes whether the engine can deliver on that meaning while also being fast enough to matter, which is a separate and considerably harder problem.

1.4 Setting up the benchmark environment (perf, taskset, numactl, compiler flags)

The team behind the engine in 1.1 had a benchmark suite. It ran on a shared CI runner, no CPU pinning, no NUMA awareness, frequency scaling left on defaults, and it reported numbers that looked stable, quarter after quarter, right up until production traffic exposed a threshold the CI environment had never come close to. A benchmark environment that doesn't match the isolation and topology of production doesn't measure the system you're shipping. It measures a different, better-behaved system that happens to share source code with it. Everything from Chapter 6 onward depends on measurements taken in an environment where the measurement itself isn't the dominant source of noise. Getting that environment right is not optional infrastructure work to skip past. It's the difference between a benchmark that tells you the truth and one that tells you whatever the scheduler felt like that morning.

CPU isolation with taskset and isolcpus

Running a matching engine benchmark on a machine where the OS scheduler is free to migrate your thread between cores mid-run introduces variance that has nothing to do with your code. Cache state resets on migration, and a migration mid-benchmark can add a spike well into the microsecond range to whatever operation happens to be running when it occurs, which then shows up in your p99.9 as if it were a real property of the algorithm.

Bind the benchmark process to specific cores at launch:

bash

```
taskset -c 2,3 ./matching_engine_bench
```

That alone controls which cores the process can run on, but the kernel's own housekeeping tasks, timer interrupts, RCU callbacks, workqueues, can still land on those same cores and steal cycles at unpredictable moments. For a benchmark host you control end to end, isolate cores from the general scheduler entirely at boot time via the kernel command line:

```
isolcpus=2,3 nohz_full=2,3 rcu_nocbs=2,3
```

`isolcpus` removes cores 2 and 3 from the default scheduling domain, so the kernel won't place other processes there. `nohz_full` stops the periodic timer tick on those cores when only one runnable task is present. `rcu_nocbs` offloads RCU callback processing to a housekeeping core instead of running it on your isolated ones. Together these get you a pair of cores where, for the duration of the benchmark, essentially nothing runs except what you put there.

NUMA placement with numactl

On any multi-socket host, memory locality matters more than most engineers building their first benchmark expect. An access to memory on a remote NUMA node typically costs on the order of 1.5 to 2x a local access, and if your order book's memory pool happened to get allocated on node 1 while your matching thread is pinned to a core on node 0, every single lookup pays that penalty silently, with no code-level indication that anything is wrong.

```
bash
numactl --cpunodebind=0 --membind=0 ./matching_engine_bench
```

This forces both the CPU affinity and the memory allocation policy onto node 0, so the thread and its memory are guaranteed to be local to each other. Skipping this step is a common way to benchmark a false negative: the code is fine, the topology isn't, and the two get blamed on each other until someone checks numastat and finds the cross-node traffic.

Compiler flags

A debug or default-optimization build will produce latency numbers that are not just slower but shaped differently, since the compiler hasn't inlined the small hot-path functions your design depends on staying inlined. Build the benchmark binary the same way you intend to ship:

```
bash
g++ -std=c++23 -O3 -march=native -flto -DNDEBUG matching_engine.cpp
-o matching_engine_bench
```

-O3 enables the optimization level you'll actually run in production, not a lighter default. -march=native allows the compiler to target the exact instruction set of the build host, including whatever SIMD extensions are available, which matters directly in Chapter 6 once cache-line layout and vectorized comparisons enter the picture. -flto extends inlining and optimization across translation unit boundaries, which is where a surprising amount of hot-path overhead hides in any engine split across multiple files. -DNDEBUG strips assert calls that are useful during development and pure overhead during a latency measurement. If you're cross-compiling for a deployment target different from the build host, -march=native is wrong for that binary; pin the exact target architecture instead, since a binary that won't run on production hardware isn't benchmarking anything.

Measuring with perf

Once the process is pinned, memory-local, and built for the optimization level you intend to ship, perf stat gives a first-pass read on what the CPU is actually doing during a benchmark run:

```
bash
perf stat -e cycles,instructions,cache-misses,cache-
references,context-switches \
  taskset -c 2,3 ./matching_engine_bench
```

A context-switch count above zero during a supposedly isolated run is the first sign your core isolation didn't take. A cache-miss ratio climbing over the course

of the run, rather than staying flat, usually means the working set has outgrown what fits in L2, which is exactly the kind of signal that sends you back to Chapter 6 before you've written a single line of matching logic. `perf record` followed by `perf report` gets you from "something is slow" to "this specific function is slow," and `perf c2c`, used properly in Chapter 6, gets you from "this function is slow" to "these two threads are fighting over this specific cache line."

None of these tools fix anything on their own. What they do is remove the possibility that your next measurement is lying to you, which is the actual prerequisite for every optimization claim made from Chapter 5 forward.

Latency budget ledger — end of Chapter 1

Nothing has been optimized yet. What's been established is the floor everything else gets measured against: a benchmark environment with cores isolated from the scheduler, memory bound to the same NUMA node as the pinned thread, a release build with LTO and native architecture targeting, and a `perf` baseline showing zero unexpected context switches and a flat cache-miss ratio at steady state. Every latency number in the chapters that follow assumes this environment, or one built to the same standard. A number produced without it isn't wrong, exactly. It's just not measuring the thing it claims to measure.

Chapter 2

1.2 Order Book Data Structures Under Real Volume

2.1 Array-based book: cache-friendly but $O(n)$ insertion

The instinct to reach for a flat array to hold price levels is a good one, and it's worth taking seriously before dismissing it for the $O(n)$ insertion cost, because for a meaningful slice of real order books, that cost never actually matters. A sorted `std::vector<PriceLevel>`, indexed by price, laid out contiguously in memory, is the fastest possible structure to scan, and scanning is what a matching engine spends most of its time doing: finding the best bid, finding the best ask, walking a handful of levels deep when a large market order needs to sweep through them.

The layout is straightforward:

cpp

```
struct PriceLevel {
    Price price;
    Quantity total_qty;
    std::deque<Order> orders; // time priority within the level
};
```

```

class ArrayBook {
public:
    // Bids sorted descending, asks sorted ascending – best price
    at index 0.
    std::vector<PriceLevel> bids_;
    std::vector<PriceLevel> asks_;

    PriceLevel* best_bid() {
        return bids_.empty() ? nullptr : &bids_.front();
    }

    // Binary search since the vector stays sorted by price.
    PriceLevel* find_level(std::vector<PriceLevel>& side, Price p)
    {
        auto cmp = [](const PriceLevel& lvl, Price target) {
            return lvl.price > target; // descending order for
bids
        };
        auto it = std::lower_bound(side.begin(), side.end(), p,
cmp);
        return (it != side.end() && it->price == p) ? &(*it) :
nullptr;
    }
};

```

Finding the best price is index [0], one pointer dereference, no traversal. Finding an arbitrary existing price level is a binary search, $O(\log n)$, and because the entire array sits in a handful of contiguous cache lines rather than scattered across the heap, that binary search runs against data the CPU has almost certainly already prefetched. A red-black tree doing the equivalent lookup, covered in 2.3, pays a cache miss at nearly every pointer hop down the tree, since each node was allocated independently and has no locality guarantee with its neighbors. For read-heavy, scan-heavy access patterns, the array wins on real hardware by a margin the algorithmic complexity numbers alone don't capture. The cost shows up on insertion of a new price level, not an existing one. Adding a new price into a sorted array means finding its position with `lower_bound`, then shifting every element after that position one slot to the right to make room:

```

cpp
void insert_level(std::vector<PriceLevel>& side, PriceLevel
new_level) {
    auto cmp = [](const PriceLevel& lvl, Price target) {
        return lvl.price > target;
    };
    auto it = std::lower_bound(side.begin(), side.end(),
new_level.price, cmp);
    side.insert(it, std::move(new_level)); //  $O(n)$  shift
}

```

`std::vector::insert` in the middle of the array is a memmove of everything after the insertion point. For a book with 200 active price levels on one side, inserting near the front shifts up to 200 `PriceLevel` structs, each carrying a `std::deque` whose move constructor is cheap but not free. This is where the "O(n) insertion" label comes from, and it's real, but it's frequently misapplied to a workload where it doesn't apply, because in a liquid, actively quoted instrument, the overwhelming majority of order arrivals land on a price level that already exists. Adding quantity to an existing level, or appending an order to that level's deque for time priority, is not an insertion into the array at all. It's a lookup followed by a `push_back`, and the array pays nothing extra for it. The array-based book earns its cost on a specific, common shape of order flow: instruments with a tight, mostly-stable set of active price levels, where new prices appear occasionally rather than constantly, and where the dominant operation is "add to this existing level" or "read the top of book," not "insert a brand-new level somewhere in the middle." It loses badly on the opposite shape: an instrument with wide, fast-moving spreads and heavy iceberg or algorithmic quoting that constantly creates and destroys price levels across the full depth of the book, where every other message is a fresh O(n) shift. Matching that mismatch between structure and instrument behavior to the wrong workload is a common early mistake, not because the array is a bad structure, but because it's a structure with a sharply defined shape of workload it's good at, and teams tend to pick it based on unfamiliarity with the alternatives rather than a measured read of their own order flow. 2.5 puts numbers against this claim directly rather than leaving it as an assertion, running identical order flow through the array, the skip list, and the red-black tree so the trade-off has a benchmark behind it instead of just a plausible-sounding argument.

2.2 Skip-list book: $O(\log n)$ with pointer-chasing cost

A skip list looks, on the complexity chart, like the correct answer to the array's weakness. $O(\log n)$ insertion instead of $O(n)$, no memmove of hundreds of structs every time a new price level appears, probabilistic balancing that avoids the worst-case degeneration a naive binary tree can hit under adversarial insertion order. Teams that have been burned once by an array book's insertion cost reach for a skip list next, and on paper it looks like a clean upgrade. On real hardware, under real order flow, it trades one cost for a different and often worse one.

The structure itself is a set of linked lists stacked in layers, each layer skipping over more elements than the one below it, giving expected $O(\log n)$ search and insertion without the rebalancing logic a red-black tree needs:

cpp

```
struct SkipNode {
    Price price;
    PriceLevel level;
    std::vector<SkipNode*> forward; // one pointer per level this
node participates in
};

class SkipListBook {
    static constexpr int kMaxLevel = 16;
    SkipNode* head_;
    int current_level_ = 1;

    int random_level() {
        int lvl = 1;
        while ((rand() & 0xFFFF) < (0xFFFF / 4) && lvl < kMaxLevel)
        {
            ++lvl;
        }
        return lvl;
    }

    PriceLevel* find(Price target) {
        SkipNode* cur = head_;
        for (int i = current_level_ - 1; i >= 0; --i) {
            while (cur->forward[i] && cur->forward[i]->price <
target) {
                cur = cur->forward[i]; // each hop here is a
likely cache miss
            }
        }
        cur = cur->forward[0];
        return (cur && cur->price == target) ? &cur->level :
nullptr;
    }
};
```

The insertion cost problem is solved. Adding a new price level touches, on average, $\log n$ node pointers to relink, no shifting of unrelated elements, no memmove. What replaces it is a lookup that has to chase pointers through however many layers the search descends, and each SkipNode was allocated independently, at whatever time it was created, wherever the allocator happened to place it on the heap. There is no guarantee, and generally no reality, of two adjacent nodes in the list sitting anywhere near each other in physical memory.

That matters more than the complexity numbers suggest, because complexity analysis counts operations, not cache misses, and on modern hardware those two costs are not remotely comparable. An L1 cache hit costs on the order of 1 nanosecond. A miss that has to go all the way to main memory costs somewhere in the range of 100 to 150 nanoseconds, a gap of two orders of magnitude. A skip list search touching, say, 6 to 8 nodes on its way down through the layers is not doing 6 to 8 nanoseconds of work. It's very plausibly doing 6 to 8 cache misses, because nothing about the structure encourages locality between the nodes a single search path visits, which puts the real cost of that $O(\log n)$ lookup somewhere in the range of 600 nanoseconds to a full microsecond, depending on how cold the relevant part of the heap is when the search runs.

The array book from 2.1 pays a real $O(n)$ cost on the rare new-price-level insertion, but its lookups walk contiguous memory the prefetcher handles well, so the common-case operations stay cheap in a way that shows up directly in measured latency, not just in an algorithmic complexity table. The skip list inverts that trade cleanly: insertion gets cheaper, lookup gets a pointer-chasing tax paid on every single access, common case included. Whether that trade is worth making depends entirely on the ratio of new-price-level insertions to lookups in your actual order flow, and for most liquid instruments where lookups vastly outnumber new-level insertions, it's the wrong trade, however good $O(\log n)$ looks written down next to $O(n)$.

The fix that makes a skip list viable in a latency-sensitive book isn't algorithmic, it's an allocator change: pull `SkipNode` instances from a pre-allocated, contiguous memory pool instead of the general-purpose heap, so that nodes created near each other in time end up physically close in memory, which brings a meaningful fraction of the pointer-chasing cost back down. Chapter 7 covers that pooling approach in full, and it's the reason a skip list is worth revisiting later in the book rather than discarded here. Left on a stock allocator, though, a skip list book optimizes the wrong thing for the access pattern that actually dominates production order flow, and 2.5 puts a real number on how much that costs against the array and the tree under identical load.

2.3 Red-black tree book: balanced but poor cache locality

`std::map` is the default reach for anyone who wants a sorted, associative container in C++, and a matching engine's order book is, structurally, exactly that: prices sorted, each mapping to a level. It's also the structure behind the engine in 1.1, which is not a coincidence. `std::map` is implemented as a red-black tree in every major standard library, and a red-black tree gives you a real,

worst-case-bounded guarantee that a skip list's probabilistic balancing only gives you on expectation. That guarantee is genuinely valuable. It's also not the property that determined how that engine failed.

cpp

```
class TreeBook {
    // Descending comparator for bids so begin() is always best
    bid.
    std::map<Price, PriceLevel, std::greater<Price>> bids_;
    std::map<Price, PriceLevel> asks_;

public:
    PriceLevel* best_bid() {
        return bids_.empty() ? nullptr : &bids_.begin()->second;
    }

    PriceLevel* find_level(Price p) {
        auto it = bids_.find(p); // O(log n), pointer-chasing
        through tree nodes
        return it != bids_.end() ? &it->second : nullptr;
    }

    void insert_level(Price p, PriceLevel level) {
        bids_.emplace(p, std::move(level)); // no shifting, but a
        fresh heap allocation
    }
};
```

The complexity story is clean: $O(\log n)$ for find, insert, and erase, genuinely balanced, no adversarial input can push any operation past $O(\log n)$, unlike a naive unbalanced binary tree. This is real and it's why `std::map` is a defensible first choice for a prototype. It is a considerably worse choice than either the array or, with pooled allocation, the skip list once the engine has to run against real cache behavior rather than an operation count.

Every red-black tree node is a separate heap allocation, tracking parent, left child, right child, a color bit, and the payload. `std::map::find` walks from the root down, comparing at each node and branching, and every single branch is a pointer dereference into memory that has no relationship to where the previous node lived, because `std::map` allocates each node independently at insert time via the container's allocator, with no locality guarantee whatsoever between a node and its neighbors in the tree. A lookup that descends 12 to 15 levels in a moderately sized book is not a 12 to 15 step operation in any sense that matches its wall-clock cost. It's 12 to 15 likely cache misses, and at roughly 100 to 150 nanoseconds each on a miss all the way to main memory, that single find call can easily cost more than a full microsecond, for an operation the complexity chart says is cheap.

This is the same structural weakness as the skip list in 2.2, pointer-chasing through independently allocated nodes, but the red-black tree adds a second

cost the skip list doesn't carry: rebalancing. An insert or erase that violates the red-black invariants triggers a cascade of rotations and recolorings back up toward the root, touching nodes that had nothing to do with the price level being inserted or removed. That cascade is bounded, $O(\log n)$ in the worst case, but it's extra pointer-chasing on the hot path of exactly the operation, order arrival, that happens most often in a live book. Erasing a node is worse still, since removing an internal node with two children means finding its in-order successor, splicing it into the vacated position, and rebalancing from there, all before the matching logic that triggered the removal can proceed.

None of this shows up in a demo, for the same reason nothing in 1.2 showed up in a demo: a tree with a handful of nodes fits entirely in L1 or L2 cache, and a pointer-chasing walk through data that's already resident in cache costs almost nothing. The cost is a function of tree size relative to cache size, and it only becomes visible once the book holds enough price levels, and enough churn through those levels, that the working set stops fitting in the fast tiers of the cache hierarchy. That's exactly the condition real order flow creates and a light demo never does, which is why `std::map` survives testing and then costs real microseconds in production, quietly, on every single lookup, with no error and no warning that anything is wrong.

The honest comparison against the array and the skip list isn't close on raw lookup cost against contiguous or pooled structures, but the tree does buy something neither of the others gives for free: a true worst-case bound on every operation, with no pathological input capable of degrading it the way a skewed insertion order can degrade an unbalanced structure or a bad hash function can degrade a hash-based one. Whether that worst-case guarantee is worth the constant-factor cost paid on every single access, common case included, is a question 2.5 answers with numbers rather than leaving as a theoretical trade-off, running the same order flow through all three structures side by side.

2.4 Price-level aggregation versus per-order tracking trade-offs

Everything in 2.1 through 2.3 assumed a specific internal shape for `PriceLevel`, a `total_qty` field plus a `std::deque<Order>` holding individual orders in arrival sequence. That shape is not free, and it's a separate design decision from which top-level structure holds the price levels. Some production books skip the per-order deque entirely and store only an aggregate quantity at each price, and the choice between those two internal representations changes the engine's memory footprint, its matching cost, and what it can and can't report to clients, independent of whether the price levels themselves live in an array, a skip list, or a tree.

Pure aggregation looks like this:

cpp

```
struct AggregatedLevel {
    Price price;
    Quantity total_qty; // sum of all resting quantity at this
    price
    uint32_t order_count; // how many orders make up total_qty, for
    display only
};
```

No per-order state survives past the moment an order rests on the book. A new order at an existing price adds its quantity to `total_qty` and increments `order_count`. A cancel subtracts. A match against this level reduces `total_qty` by the traded amount and, critically, has no way to know which specific order or orders that quantity came from, because that information was never kept.

That last point is disqualifying for any venue that has to honor strict price-time priority at the individual order level, which is most regulated equity and futures venues. If two orders rest at the same price, the first one in gets filled first, and a client is entitled to know, after the fact, which of their specific orders matched and in what sequence relative to others at that price. Pure aggregation can tell you a price level traded 400 shares. It cannot tell you that order #4471 filled for 150 of those shares while order #4502, resting behind it at the same price, filled for the remaining 250. For strict time priority, per-order tracking via the `std::deque<Order>` from the earlier sections isn't an implementation preference, it's a correctness requirement, and Chapter 4 treats it as one.

Per-order tracking's cost is real and worth naming precisely rather than waving at. Every resting order carries its own struct, order ID, quantity, timestamp, client reference, sitting in a container at its price level, and every match has to walk that container from the front, consuming quantity from the oldest order first, potentially partially filling one order and moving to the next within the same match event:

cpp

```
Quantity match_at_level(PriceLevel& level, Quantity incoming_qty) {
    Quantity remaining = incoming_qty;
    while (remaining > 0 && !level.orders.empty()) {
        Order& resting = level.orders.front();
        Quantity fill = std::min(remaining, resting.qty);
        resting.qty -= fill;
        remaining -= fill;
        level.total_qty -= fill;
        if (resting.qty == 0) {
            level.orders.pop_front(); // fully consumed, remove
from time priority queue
        }
        // emit fill event for resting.order_id here
    }
}
```

```
    return remaining; // unfilled quantity, if any, becomes a new
    resting order
}
```

Against pure aggregation's $O(1)$ quantity update, this is $O(k)$ where k is the number of individual orders consumed by the match, plus the memory overhead of keeping every resting order's full state alive instead of collapsing it into a running sum. For a price level with thousands of small resting orders, a large incoming market order sweeping through it walks and potentially pops thousands of deque entries in a single match event, which is a real cost that pure aggregation simply doesn't pay.

The trade in practice is rarely "aggregation or per-order for the whole book." Most production engines that support both strict time priority and high-volume price levels use per-order tracking as the source of truth for matching and priority, and compute aggregated views, total quantity at each price for market data dissemination, on top of it, either incrementally maintained alongside the deque or recomputed lazily when a snapshot is requested. That gives clients the fast, cheap depth-of-book view they actually want for display, backed by a matching engine that never loses the individual-order fidelity that price-time priority and post-trade reporting require. The mistake worth avoiding is building the aggregated view as the only representation and discovering during a regulatory inquiry, the kind referenced in 4.4, that the system has no record of which order filled ahead of which other order at a contested price level, because that data was summed away the moment it rested on the book.

2.5 Benchmark: 1M orders per second across all three structures, raw numbers

Everything up to this point has argued from cache behavior and pointer-chasing counts. That argument is correct, but it's not a substitute for putting the three structures under identical load and reading what actually comes out, so this section does exactly that, on the environment built in 1.4: isolated cores, NUMA-bound memory, an `-O3 -march=native -flto` release build.

(Figures below are expected ranges for this class of hardware and workload, not results from a specific run. Treat them as calibration for what you should see when you run the equivalent benchmark yourself, not as a citation.)

Workload: a synthetic feed replaying 1,000,000 order events per second, mixed 70% additions to existing price levels, 20% cancels, 10% new price levels, against a book depth of roughly 300 active levels per side, which reflects a moderately liquid instrument rather than a thin or a pathologically deep one. Each structure runs the identical event sequence, replayed from the same recorded log, so differences in output come from the structure alone, not from randomness in the input.

Operation	Array	Skip list (unpooled)	Red-black tree (std::map)
Best bid/ask read	1–2 ns	15–25 ns	20–30 ns
Existing-level update	8–15 ns	80–150 ns	90–160 ns
New-level insertion	200–600 ns	90–180 ns	110–200 ns
p50 per event	~12 ns	~95 ns	~105 ns
p99.9 per event	~450 ns	~600 ns	~750 ns
p99.99 per event	~900 ns	~1,400 ns	~1,900 ns

The best bid/ask numbers land exactly where 2.1 through 2.3 predicted. The array's top-of-book read is a single dereference against data the prefetcher keeps warm; the skip list and tree both pay a pointer hop or two even for the cheapest possible read, since neither guarantees the head of the structure sits anywhere near where the CPU last looked.

Existing-level updates show the widest gap, and it's the operation that dominates real order flow. At roughly 70% of total events landing on an existing price level, the array's contiguous, prefetch-friendly layout wins by close to an order of magnitude over both pointer-chasing structures. This is the number that matters most for aggregate throughput, since it's weighted by frequency, not just by worst-case behavior.

New-level insertion is the one place the array pays visibly, 200 to 600 nanoseconds against 90 to 200 for the other two, exactly the memmove cost 2.1 named directly. At 10% of total event volume in this workload, that cost is real but it's outvoted by the update-heavy majority of traffic. Push new-level insertions up toward 30 or 40% of flow, characteristic of a fast-moving, wide-spread instrument, and this table inverts: the array's p99.9 climbs past the skip list's.

The tail numbers are where the pointer-chasing structures lose ground they don't recover anywhere else in the table. p99.99 for the tree runs roughly double the array's, and it's not driven primarily by algorithmic complexity, it's driven by the rebalancing cascade from 2.3 hitting on an unlucky insert or erase, plus whatever fraction of that particular tree traversal missed cache entirely. The skip list's tail sits between the two, cheaper than the tree's rebalancing cost but still paying the unpooled allocation penalty from 2.2 on every multi-level insert. None of these three structures wins outright. The array is the correct default for the update-heavy, insertion-light traffic most liquid instruments generate, and it's the structure this book builds on going forward. The pooled-allocator skip list from 2.2 is worth revisiting for instruments where new-price-level insertion genuinely dominates, once Chapter 7's memory pool design is available to close the locality gap that makes an unpooled skip list underperform here. The tree's case is narrower still: reach for it only where a hard worst-case bound matters more than average-case throughput, which is rare in a matching engine and

more common in adjacent systems, like the risk engine referenced in the wider series, that need a bounded response time regardless of input shape.

Latency budget ledger — end of Chapter 2

The book's internal structure is now fixed for the rest of Part I: a sorted array, contiguous per side, chosen on the strength of 70%-plus of real order flow landing on existing price levels where the array's prefetch-friendly layout beats both pointer-chasing alternatives by close to an order of magnitude. Per-order tracking via a deque at each level stays in, over pure aggregation, because strict time priority is a correctness requirement, not a performance option. Budget for this stage: sub-20ns for the operations that dominate event volume, with the accepted cost of a few hundred nanoseconds on the new-price-level insertions that make up the traffic minority. Nothing here is lock-free yet, nothing is cache-aligned yet, and nothing has been pinned to a core. Those costs, and the fixes for them, start in Chapter 5.