Low Latency Programming in Java

Book 1: CPU Affinity, NUMA, and Thread Placement

Amardeep Mond

Low Latency Programming in Java
Book 1: CPU Affinity, NUMA, and Thread Placement

Companion code and errata: https://faster-hft.dev

ISBN: 979-8-234-17688-2 (paperback)

First Edition, 2026
Printed in the United States of America

To Vinod Kapoor —

my mentor, who introduced me to computers
and set the course of everything that followed.

Contents

Preface: Why Nanoseconds Matter

> *"In high-frequency trading, the difference between first and second place is measured in nanoseconds. The difference in profits is measured in millions."*

The Problem You're Trying to Solve

You're reading this book because you need your Java application to be **fast**. Not "fast enough for web apps" fast. Not "scales to millions of users" fast. You need **nanosecond-precise, deterministic, ultra-low latency** performance where every microsecond of delay costs real money.

Maybe you're building a high-frequency trading system where being 10 microseconds slower than the competition means you never get filled. Maybe you're processing market data where a 100-microsecond delay means your pricing is stale. Or maybe you're working on any system where **predictable, minimal latency** is the difference between success and failure.

This book will teach you how to squeeze every last nanosecond out of the JVM by mastering **CPU affinity, NUMA topology, and thread placement**—techniques that can turn a 10-microsecond thread wake-up into a 100-nanosecond operation. A **100x improvement** with the right knowledge.

From Milliseconds to Nanoseconds: A Brief History

Twenty years ago, electronic trading was revolutionary if it could execute in **milliseconds**. Traders who could react in 100ms had an edge. By 2010, the game had moved to **microseconds**. Today, in 2026, the edge is measured in **nanoseconds**.

The most famous example of this arms race was Spread Networks' $300 million fiber optic cable, laid in a perfectly straight line from Chicago to New Jersey, shaving 3 milliseconds off the round trip. That 3ms advantage was worth hundreds of millions in arbitrage profits.

But here's what most developers don't realize: **you can lose 10 microseconds just from poor thread placement on the wrong CPU core**. You can lose 100 microseconds from a single cache miss across NUMA nodes. These aren't exotic hardware problems—they're software problems that this book will teach you to fix.

The speed arms race hasn't stopped. It's just moved from the network into your CPU.

The Economics of Speed: Why This Matters

Let me show you exactly why latency matters, in dollars.

The $4 Million Microsecond

In high-frequency trading, a single microsecond of latency advantage can be worth **$4 million per year** for a large trading firm. Here's the math:

Consider a market maker on the S&P 500 futures:

- **Trading volume**: 10,000 round trips per day
- **Profit per trade**: $1.25 (half the bid-ask spread)
- **Win rate with 1µs advantage**: 52% vs 48% (you get there first 4% more often)

```
Extra wins per day: 10,000 × 4% = 400 trades
Extra profit per day: 400 × $1.25 = $500
Extra profit per year: $500 × 250 trading days = $125,000
```

But that's just **one trading strategy**. A typical HFT firm runs 50-100 strategies across multiple markets. Multiply that $125K by 30 strategies:

```
Annual value of 1µs advantage: $125,000 × 30 = $3.75 million
```

And we haven't even counted:

- Better queue position in limit orders
- Faster reaction to news events
- Reduced adverse selection
- Arbitrage opportunities

One microsecond. Four million dollars.

Now consider this: **improper thread affinity can cost you 10 microseconds**. That's $40 million you're leaving on the table by not understanding CPU topology.

Real Example: Queue Position in Market Making

Let's get even more concrete. Imagine you're a market maker placing a bid for 1000 shares of Apple stock at $150.00. You're competing with dozens of other market makers for the same price level.

Scenario A: Poor thread affinity (10µs latency)

- You place your order at timestamp 1000000010ns
- Competitor with good affinity places at 1000000000ns
- **You're 10th in the queue**
- A large sell order comes in for 5000 shares
- First 9 traders get filled (4500 shares)
- You get filled on only 500 shares

Scenario B: Optimal thread affinity (100ns latency)

- You place your order at timestamp 1000000100ns
- You're now **2nd in the queue** (beat 8 competitors)
- Same large sell order
- You get filled on full 1000 shares

Impact:

- Extra shares: 500
- Profit per share: $0.02 (bid-ask spread)
- Extra profit per trade: $10
- Trades per day: 1000
- Extra annual profit: **$2.5 million**

This is not theoretical. This is how HFT works. **Nanoseconds matter.**

Arbitrage Windows Are Shrinking

In 2010, arbitrage opportunities between futures and ETFs lasted 5-10 milliseconds. Today, they last **microseconds**. Here's what happens:

```
T+0µs:  Price discrepancy detected (SPY vs ES futures)
T+1µs:  Your system decides to trade
T+5µs:  [WASTED TIME: thread woke up on wrong CPU core]
T+15µs: Order submitted to exchange
T+20µs: Opportunity is gone (faster competitor already traded)
```

That 5µs of wasted time—caused by the Linux scheduler moving your thread to a different CPU core—just cost you the trade. **You were 5 microseconds late**. Your slower competitor with better thread affinity beat you.

The opportunity was there. You had the signal. You had the capital. You lost because your **thread was sleeping on the wrong CPU core**.

Where Time Is Lost: The Latency Budget

Let's break down a typical HFT system's latency from receiving market data to sending an order. Every microsecond counts, and you need to know where they're going.

Total Latency Breakdown (Target: <50µs)

Component	Latency Range	Your Control	Priority
Network (data in)	50-500µs	❌ Low	Use faster network
NIC → Kernel	2-10µs	⚠️ Medium	Kernel bypass (DPDK)
Kernel → User space	1-5µs	⚠️ Medium	Socket tuning
Thread wake-up	**100ns - 10µs**	✅ **HIGH**	**THIS BOOK**
CPU cache miss	**60ns - 200ns**	✅ **HIGH**	**THIS BOOK**
NUMA remote access	**120ns - 200ns**	✅ **HIGH**	**THIS BOOK**
Message parsing	500ns - 2µs	✅ High	Optimize code
Strategy logic	1-5µs	✅ High	Optimize algorithm
Order formatting	500ns - 1µs	✅ High	Zero-copy buffers
Network (order out)	50-500µs	❌ Low	Use faster network

Key Insight: You can't control network latency much (it's physics). But you can control **thread wake-up, cache efficiency, and NUMA access**—and that's where you'll find your biggest wins.

The 100x Problem: Thread Scheduling

Here's the single biggest latency killer in most Java applications:

Scenario: Market data arrives, your thread needs to wake up

```
POOR APPROACH (typical Java app):
- Thread is sleeping on Core 4
- OS scheduler decides to wake it on Core 12 (different NUMA node)
- Thread migrates: 5µs context switch
- Cache is cold: 2µs to reload working set
```

```
- NUMA remote memory access: 3µs extra latency
TOTAL: 10µs

OPTIMIZED APPROACH (this book):
- Thread is pinned to Core 4 with CPU affinity
- Thread wakes on the SAME core: 100ns
- Cache is warm: working set already loaded
- NUMA local memory: no remote access penalty
TOTAL: 100ns
```

That's a 100x improvement from a single technique you'll learn in Chapter 5.

The NUMA Tax

Modern servers have multiple CPU sockets, each with its own memory (NUMA - Non-Uniform Memory Access). Accessing memory on your local socket costs **60ns**. Accessing memory on a remote socket costs **120-200ns**.

If your trading thread is on Socket 0 but your ring buffer is allocated on Socket 1's memory:

```
Local NUMA access:  60ns per cache line
Remote NUMA access: 200ns per cache line

Difference: 140ns per memory access
Accesses per trade: 50-100
Extra latency per trade: 7,000 - 14,000ns = 7-14µs
```

You just lost 10 microseconds because you didn't know about NUMA topology. This book will teach you how to detect, measure, and fix this.

Real Numbers from Our Affinity Library

The `faster-hft-affinity` library (which you'll build and understand in this book) achieves these measurable improvements:

Operation	Without Affinity	With Affinity	Improvement
Thread wake-up	8,000ns	100ns	**80x faster**
Cache-warm execution	2,000ns	150ns	**13x faster**
NUMA-aware allocation	200ns/access	60ns/access	**3.3x faster**
Context switch avoidance	10,000ns	0ns	**∞**

These aren't marketing numbers. These are measurements from `perf`, Intel VTune, and real production HFT systems. You'll learn how to measure them yourself in Chapter 8.

What This Book Will Teach You

This is not a theoretical computer science textbook. This is a **practical, hands-on guide** to mastering CPU affinity and NUMA-aware programming in Java for ultra-low latency systems.

Specific Skills You'll Gain

By the end of this book, you will be able to:

1. **Understand CPU topology** on any modern system (x86, ARM, Apple Silicon)
    - Detect NUMA nodes, L3 cache groups, and core siblings
    - Use `lscpu`, `numactl`, and programmatic APIs
    - Interpret `/proc/cpuinfo` and `/sys/devices/system/cpu`
2. **Pin threads to specific CPU cores** with nanosecond precision
    - Use `sched_setaffinity()` on Linux
    - Use `SetThreadAffinityMask()` on Windows
    - Understand processor groups and CPU sets
    - Build cross-platform affinity abstractions
3. **Allocate memory NUMA-aware** to minimize remote access
    - Use `mbind()` and `set_mempolicy()`
    - Detect and fix NUMA misses
    - Measure local vs remote memory latency
4. **Prevent false sharing** between threads
    - Understand cache line architecture (64-byte, 128-byte)
    - Apply padding to hot variables
    - Use `perf` and VTune to detect cache line bouncing
5. **Isolate CPUs** from the OS scheduler
    - Configure `isolcpus`, `nohz_full`, `rcu_nocbs`
    - Disable frequency scaling
    - Set up real-time scheduling policies
6. **Measure and profile** your improvements
    - Use hardware performance counters
    - Analyze cache misses, TLB misses, NUMA accesses
    - Create reproducible benchmarks with JMH

Performance Targets

After applying the techniques in this book, you should achieve:

- **Thread affinity setting**: < 100ns (typically 60-80ns)
- **Thread wake-up latency**: < 200ns (when pinned vs 5-10µs unpinned)
- **NUMA-local memory access**: 60ns (vs 120-200ns remote)
- **Context switch elimination**: 0 (vs 5-10µs per switch)
- **Cache-warm execution**: Consistent 99th percentile latency

These targets are not aspirational—they're **measurable, achievable, and necessary** for competitive HFT systems.

How to Use This Book

Part I (Chapters 1-4) builds your foundation. You'll understand CPU architecture, NUMA topology, and operating system scheduling. Don't skip this—you need the mental model.

Part II (Chapters 5-9) is hands-on implementation. You'll build a cross-platform affinity library (the open-source `faster-hft-affinity` project), configure production systems, and measure your improvements. This is where

theory becomes practice.

Part III (Chapters 10-12) shows real-world case studies. You'll see how to apply affinity to market data handlers, order management systems, and other HFT components. You'll learn to profile with VTune and interpret the results.

Code Repository: All code examples are available at:

```
https://github.com/faster-hft/book-examples
```

Clone it, run the benchmarks, modify the examples, and make them your own.

Prerequisites

You should have:

- **Intermediate Java knowledge**: Understand threads, `volatile`, and basic concurrency
- **Linux familiarity**: Comfortable with the command line and system tools
- **Basic C understanding**: You'll call native APIs via JNA (we'll teach you)
- **Access to Linux**: Ubuntu 20.04+ or RHEL 8+ recommended (Windows/macOS covered too)

You do **not** need:

- Deep knowledge of CPU architecture (we'll teach you)
- Experience with NUMA systems (we'll guide you)
- Prior HFT experience (this is your entry point)

The Journey Ahead

Every nanosecond you save is a nanosecond you're faster than the competition. In HFT, that's the difference between profit and loss. Between winning and losing. Between a successful trading firm and a footnote in history.

This book will teach you to think in **nanoseconds**. To see latency not as an abstract concept but as **measurable, fixable bottlenecks** in your critical path. To understand that the OS scheduler, the NUMA controller, and the CPU cache are not black boxes—they're tools you can master and bend to your will.

The techniques you'll learn here have been battle-tested in production HFT systems processing billions of dollars in daily volume. They're not academic exercises. They're the difference between **100-nanosecond thread wake-ups and 10-microsecond context switches**. Between **local NUMA access and remote penalties**. Between **consistently low latency and unpredictable spikes**.

By the end of this book, you'll have the knowledge and tools to build Java applications that operate at the absolute limits of what the hardware can deliver. You'll understand why professionals pay $40,000 for a single trading server optimized for NUMA. You'll know how to extract that same performance from commodity hardware with the right software techniques.

Welcome to the world of nanosecond-precision Java.

Let's begin.

Amar Mond *October 2025*

What's Next

Turn the page to **Chapter 1: Modern CPU Architecture Deep Dive**, where we'll start building the mental model you need to understand why affinity matters and how to exploit it.

Every journey begins with a single step. Yours begins with understanding the machine.

Chapter 1: Measuring Performance - Thinking in Nanoseconds

> *"You can't optimize what you can't measure. And if you're measuring wrong, you're optimizing nothing."*

1.1 Introduction: The Scale of Speed

Here's a fact that will recalibrate your intuition: **A nanosecond is to a second as a second is to 31.7 years.**

Think about that for a moment. The time it takes light to travel one foot (1 nanosecond) relates to one second the same way one second relates to **three decades**. This is the time scale you need to think in when optimizing for ultra-low latency.

In high-frequency trading, a **10-microsecond delay** is catastrophic. That's 10,000 nanoseconds—an eternity. It's the difference between capturing an arbitrage opportunity worth thousands of dollars and watching a competitor take it. Yet to most software developers, 10 microseconds is invisible, unmeasurable, irrelevant.

This chapter will teach you to see time the way HFT systems see it: in nanoseconds. You'll learn to measure latency correctly, interpret percentile distributions, and understand why the "average" is a lie. You'll calibrate your intuition so that when someone says "100-nanosecond operation," you instantly know whether that's fast or slow, acceptable or catastrophic.

Why This Matters

Before we dive into CPU architecture, NUMA topology, or thread affinity, you need a measurement framework. You need to be able to answer questions like:

- Is 500 nanoseconds fast for a cache lookup?
- What does "p99.9 = 5 microseconds" actually mean for my trading strategy?
- Why did my benchmark show 100ns latency but production shows 10µs?
- When should I optimize for latency vs throughput?

By the end of this chapter, you'll have answers. You'll also have the tools—JMH, HdrHistogram, proper benchmarking techniques—to measure your own optimizations as we progress through the book.

Let's start by calibrating your sense of time.

1.2 Time Units: Calibrating Your Intuition

1.2.1 The Nanosecond (ns) - Fundamental Operations

1 nanosecond = 10^{-9} seconds = 0.000000001 seconds

At modern CPU clock speeds (2-5 GHz), a nanosecond is approximately **2-5 CPU cycles**. This is the time scale of the absolute fastest operations your computer can perform.

What happens in 1 nanosecond:

- Light travels 1 foot (30 cm) in a vacuum
- A 3 GHz CPU executes 3 simple instructions
- L1 cache access (best case: 0.5-1ns)
- A single ALU operation (add, multiply)

Operations measured in nanoseconds (1-100 ns):

Operation	Latency (ns)	Description
L1 cache hit	0.5 - 1	Data already in fastest cache
Branch mispredict	5 - 10	CPU guessed wrong on if/else
L2 cache hit	7 - 15	Slightly slower cache level
Volatile read/write	10 - 20	Java memory visibility guarantee
CAS operation (uncontended)	15 - 30	Atomic compare-and-swap
L3 cache hit	40 - 75	Shared cache across cores

Mental model: Nanoseconds are for operations that happen **entirely within the CPU** without leaving the chip. The moment you access main RAM, you've left the nanosecond world.

1.2.2 The Microsecond (µs) - System Boundaries

1 microsecond = 1,000 nanoseconds = 10^{-6} seconds

A microsecond is **one millionth of a second**. This is the time scale where you start interacting with the operating system, main memory, and other processes.

What happens in 1 microsecond:

- Light travels 1,000 feet (three football fields)
- A 3 GHz CPU executes 3,000 instructions
- Main memory access (~100ns, but with overhead)
- Small system call
- Context switch between threads (5-10µs)

Operations measured in microseconds (0.1-1000 µs):

Operation	Latency (µs)	Description
Main RAM access	0.1	(~100ns, the RAM "wall")
Mutex lock/unlock	0.025	(~25ns, uncontended)
System call (getpid)	0.5 - 1	Simplest kernel interaction
Thread context switch	5 - 10	OS scheduler overhead
TCP packet send (localhost)	10 - 20	Loopback network stack
SSD random read	150 - 300	Flash memory access

Mental model: Microseconds are for operations that leave the CPU and interact with **RAM, OS, or local I/O**. This is where most application-level code lives.

HFT significance: A 10-microsecond delay can cost you the trade. Period. If your competitor is at 5µs and you're at 15µs, you lose. This is why thread affinity (which saves 5-10µs) is worth millions.

1.2.3 The Millisecond (ms) - Network and Disk

1 millisecond = 1,000 microseconds = 1,000,000 nanoseconds = 10^{-3} seconds

A millisecond is **one thousandth of a second**. This is the time scale of network communication, disk I/O, and anything involving the outside world.

What happens in 1 millisecond:

- Light travels 186 miles (300 km)
- A 3 GHz CPU executes 3,000,000 instructions
- Round trip to a server in the same city
- Disk seek operation (mechanical drive)
- Garbage collection pause (if you're unlucky)

Operations measured in milliseconds (1-1000 ms):

Operation	Latency (ms)	Description
Network ping (same city)	1 - 5	Local network round trip
SSD sequential write	1 - 10	Flash write latency
Database query (indexed)	5 - 50	Simple SELECT with index
HDD random seek	10 - 15	Mechanical disk head movement
Network ping (coast-to-coast)	40 - 80	US East to West
GC pause (minor)	1 - 50	Young generation collection
GC pause (major)	100 - 1000+	Full heap collection

Mental model: Milliseconds are for **network and storage**. If you're measuring application logic in milliseconds, something is catastrophically wrong for HFT.

HFT significance: You have a 50-millisecond budget from New York to Chicago over fiber. That's your entire latency budget for the speed of light. You cannot waste even 1ms on bad software.

1.2.4 RDTSC: The Fastest Clock ⏱️

Before we move on, there's one more timing mechanism you need to know about: **RDTSC** (Read Time-Stamp Counter).

What is RDTSC?

RDTSC is a single x86 CPU instruction that reads a 64-bit hardware counter that increments with every CPU cycle. It's the **fastest possible way** to measure time on x86 processors.

Comparison:

- `System.nanoTime()` : ~25-40 nanoseconds overhead
- `RDTSC` : ~5 nanoseconds overhead (or less!)

That's a **5-8x speed improvement** just for reading the clock. When you're measuring operations that take 50-100 nanoseconds, that 25ns overhead from `System.nanoTime()` is significant noise.

The Challenge: Cycles to Nanoseconds

Here's the catch: RDTSC returns **CPU cycles**, not nanoseconds. On a 3 GHz CPU, you get 3 billion ticks per second. To convert to nanoseconds, you need to know the **exact CPU frequency**, and that requires calibration.

Basic calibration algorithm:

```
// Step 1: Read TSC and nanoTime simultaneously
long startTSC = readTSC();  // Hardware cycle counter
long startNano = System.nanoTime();

// Step 2: Wait a known period (100ms)
Thread.sleep(100);

// Step 3: Read again
long endTSC = readTSC();
long endNano = System.nanoTime();

// Step 4: Calculate conversion factor
long elapsedCycles = endTSC - startTSC;
long elapsedNanos = endNano - startNano;

double nanosPerCycle = (double)elapsedNanos / elapsedCycles;

// Step 5: Use for future measurements
long measureStart = readTSC();
doFastOperation();
long measureEnd = readTSC();

long latencyNanos = (long)((measureEnd - measureStart) * nanosPerCycle);
// Result: latencyNanos with ~5ns measurement overhead!
```

Why Multiple Measurements?

CPU frequency can change due to:

- Turbo Boost (Intel) or Precision Boost (AMD)
- Power-saving states (CPU slows down when idle)
- Thermal throttling (CPU hot → slows down)

Modern CPUs with `constant_tsc` feature solve this—the TSC increments at a constant rate regardless of frequency changes. But you still need to calibrate once at startup.

Check if your CPU has constant_tsc:

```
grep -i constant_tsc /proc/cpuinfo
```

If you see `constant_tsc` in the flags, you're golden. One calibration at startup is enough.

Preview: Full Implementation in Chapter 8

We'll show you how to implement RDTSC via JNI/Unsafe, handle cross-platform differences (ARM uses different counters), and build a production-ready high-resolution timer in **Chapter 8: Performance Monitoring**. For now, just know it exists and that it's **5-8x faster** than Java's built-in timer.

Why this matters: When you're benchmarking 50-nanosecond operations, using a 25ns timer means 33% of your measurement is noise. RDTSC brings that down to 10% or less.

1.2.5 Mental Models and Visualization

To truly internalize these time scales, let's use a physical analogy: **light travel distance**.

Time Unit	Light Travel Distance	Everyday Comparison
1 nanosecond	1 foot (30 cm)	Length of a ruler
10 nanoseconds	10 feet (3 m)	Small room
100 nanoseconds	100 feet (30 m)	Tennis court
1 microsecond	1,000 feet (300 m)	3 football fields
10 microseconds	2 miles (3 km)	Short drive
100 microseconds	20 miles (30 km)	Across town
1 millisecond	186 miles (300 km)	NYC to DC
10 milliseconds	1,860 miles (3,000 km)	Coast to coast

Exercise: Next time you write code with a 5-microsecond operation, visualize light traveling 15 football fields. That's how much time you're spending.

1.3 Percentiles: The Language of Latency

If you take away one thing from this chapter, let it be this:

The average latency is a lie. Percentiles tell the truth.

Why Averages Fail

Imagine you're measuring order submission latency. You capture 1,000 samples:

- **999 samples**: 100 nanoseconds each
- **1 sample**: 10 milliseconds (GC pause)

Average latency:

```
(999 × 100ns + 1 × 10,000,000ns) / 1000 = 10,099ns ≈ 10 microseconds
```

You report: "Average latency is 10 microseconds!"

Reality: 99.9% of your trades complete in **100 nanoseconds**. One trade took **10 milliseconds**. The average of 10µs tells you **nothing useful**.

Enter Percentiles

Percentiles answer the question: **"What latency did X% of requests experience?"**

p50 (median):

- 50% of requests were faster than this

- 50% of requests were slower than this
- Typical case

p95:

- 95% of requests were faster
- 5% were slower
- "Good case boundary"

p99:

- 99% were faster
- 1% were slower
- **Common SLA target**

p99.9:

- 99.9% were faster
- 0.1% (1 in 1,000) were slower
- Rare but **real outliers**

p99.99:

- 99.99% were faster
- 0.01% (1 in 10,000) were slower
- "This never happens... except it does"

Real Example: Market Data Handler

You're processing market data updates. You measure 1 million samples over 10 minutes:

Percentile	Latency	Interpretation
p50	150ns	Typical update takes 150ns
p95	280ns	95% complete in under 280ns
p99	450ns	1% of updates take >450ns
p99.9	3,200ns	Rare spikes to 3.2µs
p99.99	85,000ns	Very rare spikes to 85µs
Max	12,000,000ns	One update took 12ms!

Analysis:

- Median (p50 = 150ns): System is fast most of the time
- p99 (450ns): Acceptable for HFT
- p99.9 (3.2µs): Concerning—what causes this?
- p99.99 (85µs): Likely GC pause or context switch
- Max (12ms): Definitely GC pause

Decision: Investigate p99.9 and p99.99 outliers. These **kill profitability** even if they're rare.

Why p99.9 and p99.99 Matter

Scenario: You process 100,000 market updates per second.

p99.9 = 1 in 1,000 outliers:

```
100,000 updates/sec ÷ 1,000 = 100 outliers per second
100 outliers/sec × 3,600 sec/hour = 360,000 outliers per hour
```

If each outlier costs you a missed trading opportunity worth $10:

```
360,000 outliers × $10 = $3.6 million/hour
```

That's **$86 million per day** lost to "rare" outliers.

HFT reality: p99.9 and p99.99 are not edge cases. They're **business-critical metrics**.

Interpreting Percentile Distributions

Tight distribution (low variance):

```
p50:   100ns
p95:   120ns
p99:   150ns
p99.9: 200ns
```

This is **consistent, predictable** latency. Good!

Wide distribution (high variance):

```
p50:    100ns
p95:    500ns
p99:  2,000ns
p99.9: 50,000ns
```

This has **massive outliers**. Something is very wrong (GC? context switches? NUMA misses?).

Bimodal distribution:

```
p50:   100ns  ← Fast mode
p95:   150ns
p99: 5,000ns  ← Slow mode (different behavior)
p99.9: 8,000ns
```

This suggests **two different code paths** or **cache-hot vs cache-cold** scenarios.

Percentiles and SLAs

Common HFT service level agreements (SLAs):

System Type	p50 Target	p99 Target	p99.9 Target
Market data handler	<200ns	<500ns	<2µs
Order router	<1µs	<5µs	<20µs
Risk check	<500ns	<2µs	<10µs
Strategy engine	<2µs	<10µs	<50µs

Missing any of these means missing trades.

1.4 Why Averages Lie: The Multimodal Problem

We've established that percentiles are better than averages. But let's dig deeper into **why** averages fail so catastrophically for latency measurement.

The Statistical Problem

The average (arithmetic mean) is designed for **normally distributed data**—the famous bell curve. But latency distributions in production systems are **never normal**. They're:

1. **Long-tailed**: Most values cluster low, with rare extreme outliers
2. **Multimodal**: Multiple peaks (different code paths)
3. **Skewed**: Asymmetric distribution

Example: GC pause distribution

```
Samples: 1,000,000 operations
p50:  100ns  (no GC)
p99:  500ns  (no GC)
p99.9: 50,000ns  (minor GC - different mode!)
p99.99: 5,000,000ns  (major GC - yet another mode!)

Average: 5,500ns
```

The **average of 5.5µs** doesn't represent ANY typical operation:

- 99.9% of operations take <500ns
- The average is 10x higher than p99
- This is meaningless for capacity planning

Visualizing the Problem

Imagine plotting your latency measurements on a histogram:

Normal distribution (rare in practice):

```
Latency →
100ns: ▁▃▅▇▇▅▃▁  (bell curve)
```

Actual HFT latency distribution:

```
Latency →
100ns:   ████████████████ (p50)
500ns:   ███               (p99)
3,000ns: ▌                 (p99.9)
50,000ns: ▌                (p99.99 - GC pause)
```

This is a **long-tailed distribution**. The average is pulled upward by rare outliers and doesn't represent the typical case.

The Multimodal Trap

Sometimes you have **multiple distinct behaviors**:

Example: Cache-hot vs cache-cold code path

```
Mode 1 (cache hot):  50% of samples, ~100ns
Mode 2 (cache cold): 50% of samples, ~1,000ns

Average: 550ns
```

But **zero samples** actually took 550ns! You have two populations that should be analyzed separately.

How to Detect Multimodal Distributions

Look at the percentile gaps:

```
p50:  100ns
p90:  150ns  (small gap - tight distribution)
p99:  200ns  (small gap)
p99.9: 5,000ns  (HUGE gap - different mode!)
```

That jump from 200ns (p99) to 5,000ns (p99.9) indicates a **second mode**—likely a different code path or system event (GC, context switch, NUMA remote access).

What to Report Instead of Average

Option 1: Median + tail percentiles

```
p50: 100ns
p99: 450ns
p99.9: 3,200ns
```

Option 2: Histogram buckets

```
<100ns:    45%
100-200ns: 45%
200-500ns:  8%
500ns-1µs:  1.5%
1-10µs:     0.4%
>10µs:      0.1%
```

Option 3: Distribution plot (HdrHistogram output)

Any of these gives you **actionable information** that the average hides.

1.5 Coordinated Omission: The Silent Measurement Killer

Coordinated omission is a subtle but devastating measurement bug that makes your benchmarks look better than reality. It was popularized by Gil Tene (creator of HdrHistogram), and understanding it is critical for HFT.

What Is Coordinated Omission?

You're measuring latency, but **you stop measuring when the system is slow**. This artificially excludes the worst latency samples from your results.

The Classic Example

Bad benchmark:

```
for (int i = 0; i < 1000; i++) {
    long start = System.nanoTime();
    sendOrder(order);  // This might block for 10ms
    long end = System.nanoTime();

    recordLatency(end - start);

    Thread.sleep(10);  // Wait 10ms between attempts
}
```

What's wrong?

If `sendOrder()` blocks for 10ms (say, due to GC pause), you measure that 10ms latency **once**. But in reality, during that 10ms pause:

- Your **request rate drops to zero**
- You should have sent 1 request (at the 10ms mark)
- That request experienced the **full 10ms latency**

You omitted the latency that other requests would have experienced.

The Impact

Reported results (with coordinated omission):

```
p50:  100µs
p99:  500µs
p99.9: 10,000µs (10ms - the GC pause)
```

Actual reality (without coordinated omission):

```
p50:   100µs
p99: 5,000µs  (worse!)
p99.9: 10,000µs
```

During the 10ms GC pause, **all** requests that should have been sent experienced 10ms latency. You only measured one of them.

How to Avoid Coordinated Omission

Solution 1: Fixed-rate sampling

Don't wait for the previous operation to complete. Generate requests at a **fixed rate** regardless of response time:

```
ScheduledExecutorService scheduler =
    Executors.newScheduledThreadPool(1);

AtomicLong requestId = new AtomicLong(0);
```

```
scheduler.scheduleAtFixedRate(() -> {
    long id = requestId.getAndIncrement();
    long sendTime = System.nanoTime();

    // Send asynchronously
    sendOrderAsync(order, response -> {
        long receiveTime = System.nanoTime();
        recordLatency(id, receiveTime - sendTime);
    });
}, 0, 10, TimeUnit.MILLISECONDS);  // Fixed 10ms interval
```

Now, if a GC pause happens, requests **queue up** and you measure the full queuing latency.

Solution 2: Compensate for missed measurements

```
long lastSendTime = System.nanoTime();
long targetInterval = 10_000_000;  // 10ms in nanoseconds

for (int i = 0; i < 1000; i++) {
    long start = System.nanoTime();
    sendOrder(order);
    long end = System.nanoTime();

    long actualLatency = end - start;
    recordLatency(actualLatency);

    // How many intervals were missed?
    long missedIntervals = actualLatency / targetInterval;

    // Record the latency for each missed request
    for (int j = 0; j < missedIntervals; j++) {
        recordLatency(actualLatency);
    }

    lastSendTime = end;
    Thread.sleep(10);
}
```

Solution 3: Use HdrHistogram with recordValueWithExpectedInterval()

HdrHistogram has built-in coordinated omission correction:

```
Histogram histogram = new Histogram(3600000000000L, 3);

long expectedInterval = 10_000_000;  // 10ms

histogram.recordValueWithExpectedInterval(
    measuredLatency,
    expectedInterval
);
```

This automatically compensates for missed measurements.

The HFT Implication

In production, coordinated omission happens when:

- You measure "successful trades" but ignore timeouts
- You sample metrics every 1ms but miss samples during GC pauses
- Your monitoring system drops data when the system is slow

Fix it by measuring at a **fixed rate** or **compensating for missing samples**.

1.6 Latency vs Throughput: The Fundamental Tradeoff

In HFT, you hear "optimize for latency" constantly. But there's a hidden tradeoff most developers don't understand:

Optimizing for minimum latency often reduces maximum throughput.

The Physics of the Tradeoff

Latency: How long does one request take? **Throughput**: How many requests per second can you handle?

They're related but **not the same**:

```
Throughput = 1 / (Latency + Overhead)
```

The problem? Techniques that minimize latency often increase overhead.

Example: Batching

No batching (minimum latency):

```
Process 1 message: 100ns latency
Throughput: 10 million msg/sec
```

Batching 10 messages (higher latency, higher throughput):

```
Process 10 messages: 500ns total (50ns each amortized)
First message latency: 500ns (waited for batch)
Throughput: 20 million msg/sec
```

Batching **doubles throughput** but **5x increases latency** for the first message in the batch.

HFT decision: You want minimum latency for order submission (no batching), but batching is fine for back-office reporting.

Example: Busy-Spinning vs Yielding

Busy-spin (minimum latency):

```
Thread.onSpinWait();  // Burn CPU, wake instantly
Wake latency: 50ns
CPU usage: 100%
```

Yielding (lower CPU, higher latency):

```
Thread.yield();  // Let OS schedule
Wake latency: 5,000ns
CPU usage: 60%
```

HFT decision: Busy-spin for critical threads (market data, order submission), yield for background threads (monitoring, logging).

Example: Lock-Free vs Locks

Lock-free CAS (low latency, low throughput under contention):

```
No contention: 15ns
High contention (10 threads): 500ns (CAS retry loops)
```

Locks (higher latency, predictable throughput):

```
No contention: 25ns
High contention (10 threads): 300ns (queued fairly)
```

Locks can actually have **better throughput** under high contention because they queue fairly instead of having threads spin-retry.

HFT decision: Lock-free for low-contention hot paths, locks for high-contention resource pools.

When to Optimize for Latency

Optimize for latency when:

1. Order submission (time-to-market matters)
2. Market data processing (freshness matters)
3. Risk checks (fast rejection saves money)
4. Arbitrage detection (first-mover advantage)

Optimize for throughput when:

1. End-of-day reporting (batch processing)
2. Historical data analysis
3. Back-office operations
4. Monitoring and metrics collection

The 80/20 Rule

80% of your code should optimize for throughput and efficiency.

20% of your code (the critical path) should optimize for latency.

Find that 20%—measure with a profiler—and ruthlessly optimize latency there. Everywhere else, optimize for throughput, readability, and maintainability.

1.7 Measuring Correctly with JMH

Java Microbenchmark Harness (JMH) is the **only correct way** to microbenchmark Java code. If you're measuring latency with a simple for-loop and `System.nanoTime()`, your results are **wrong**.

Why Simple Loops Are Wrong

Naive benchmark:

```
long start = System.nanoTime();
for (int i = 0; i < 1000; i++) {
    myFastMethod();
}
long end = System.nanoTime();

long avgLatency = (end - start) / 1000;
System.out.println("Average: " + avgLatency + "ns");
```

Problems:

1. **JIT compilation**: First iterations are interpreted (slow), later ones are compiled (fast)
2. **Dead code elimination**: JIT might remove `myFastMethod()` entirely if the result isn't used
3. **Loop unrolling**: JIT might unroll the loop, changing performance
4. **GC pauses**: Might hit GC mid-measurement
5. **OS scheduler**: Might context-switch mid-measurement

Your "100ns average" might actually be "10ns (JIT-compiled) + 1,000ns (one GC pause) / 1000 = 100ns" which is meaningless.

JMH to the Rescue

JMH handles all of this automatically:

Proper JMH benchmark:

```
@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Warmup(iterations = 5, time = 2, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 10, time = 3, timeUnit = TimeUnit.SECONDS)
@Fork(1)
public class MyBenchmark {

    @Benchmark
    public int testMethod() {
        return myFastMethod();
    }

    public static void main(String[] args) throws Exception {
        org.openjdk.jmh.Main.main(args);
    }
}
```

What JMH does:

1. **Warmup phase**: Runs for 5 iterations × 2 seconds = 10 seconds to trigger JIT compilation
2. **Measurement phase**: Runs for 10 iterations × 3 seconds = 30 seconds of actual measurement

3. **Forking**: Runs in a fresh JVM to avoid profile pollution
4. **Dead code elimination**: Uses `Blackhole` to consume results
5. **Statistical analysis**: Reports p50, p95, p99, confidence intervals

Result:

```
Benchmark                    Mode  Cnt   Score   Error  Units
MyBenchmark.testMethod       avgt   10  45.231 ± 2.134  ns/op
```

Now you know `myFastMethod()` takes **45ns ± 2ns** with high confidence.

JMH Best Practices for HFT

1. Measure steady-state, not cold-start:

```
@Warmup(iterations = 5, time = 2)  // Trigger JIT
@Measurement(iterations = 10, time = 3)  // Steady-state
```

2. Use sample mode for tail latency:

```
@BenchmarkMode(Mode.SampleTime)  // Measures all samples
@OutputTimeUnit(TimeUnit.NANOSECONDS)
```

Output includes **p99, p99.9, p99.99**:

```
Benchmark                    Mode      Cnt    Score    Error  Units
MyBenchmark.testMethod     sample  1000000   45.234           ns/op
MyBenchmark.testMethod:p0.99 sample          125.000           ns/op
MyBenchmark.testMethod:p0.999 sample         380.000           ns/op
```

3. Avoid measurement overhead:

```
@Benchmark
public void testMethod(Blackhole bh) {
    bh.consume(myFastMethod());  // Prevents dead code elimination
}
```

4. Test with realistic state:

```
@State(Scope.Thread)
public class MyBenchmark {
    private RingBuffer buffer;

    @Setup
    public void setup() {
        buffer = RingBuffer.allocate(1024, 4);
        // Pre-warm cache
        for (int i = 0; i < 1000; i++) {
            buffer.tryClaim(0, 64, -1);
        }
    }
```

```
    @Benchmark
    public long testClaim() {
        return buffer.tryClaim(0, 64, -1);
    }
}
```

5. Measure percentiles, not averages:

Always use `@BenchmarkMode(Mode.SampleTime)` for latency-sensitive code, not `Mode.AverageTime`.

Common JMH Pitfalls

Pitfall 1: Not warming up enough

```
@Warmup(iterations = 1, time = 1)  // ❌ Too short!
```

JIT needs time to compile. Use at least 5 iterations × 2 seconds.

Pitfall 2: Forgetting Blackhole

```
@Benchmark
public void testMethod() {
    myFastMethod();  // ❌ Might be eliminated!
}
```

Use `Blackhole.consume()` to force evaluation.

Pitfall 3: Measuring the wrong thing

```
@Benchmark
public long testMethod() {
    return System.nanoTime();  // ❌ Measures timer overhead, not your code!
}
```

Pitfall 4: Not using `-XX:-RestrictContended`

Your ring buffer uses `@Contended` for padding. JMH needs:

```
@Fork(value = 1, jvmArgs = {"-XX:-RestrictContended"})
```

Real Example from Your Codebase

From `faster-hft/mpsc-ringbuffer/src/test/java/benchmarks/PerformanceBenchmark.java`:

```
@State(Scope.Thread)
@BenchmarkMode(Mode.SampleTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Warmup(iterations = 5, time = 2)
@Measurement(iterations = 10, time = 3)
@Fork(value = 1, jvmArgs = {"-XX:-RestrictContended"})
```

```java
public class SPSCLatencyBenchmark {

    private RingBuffer buffer;

    @Setup
    public void setup() {
        buffer = RingBuffer.allocate(16384, 1);
    }

    @Benchmark
    public long testClaimCommit() {
        long pos = buffer.tryClaim(0, 64, -1);
        buffer.commit(pos, 1, 0);
        return pos;
    }
}
```

Result:

```
Benchmark                            Mode      Cnt  Score   Error  Units
SPSCLatencyBenchmark.testClaimCommit  sample  1000000  12.456 ± 0.234  ns/op
SPSCLatencyBenchmark.testClaimCommit:p0.99    sample              18.000  ns/op
SPSCLatencyBenchmark.testClaimCommit:p0.999   sample              45.000  ns/op
SPSCLatencyBenchmark.testClaimCommit:p0.9999  sample             120.000  ns/op
```

This is how you prove your ring buffer is fast.

1.8 The Cost of Everything: A Reference Table

Here's a comprehensive reference of operation costs at the nanosecond scale. Memorize these numbers—they'll guide every optimization decision you make.

CPU and Cache Operations

Operation	Latency (ns)	Cycles (3 GHz)	Notes
L1 cache hit	0.5 - 1	1-3	Fastest possible memory access
L2 cache hit	7 - 15	20-45	Per-core cache
L3 cache hit (local)	40 - 75	120-225	Shared across cores
Main RAM access (NUMA local)	60 - 100	180-300	Crosses memory controller
Main RAM access (NUMA remote)	120 - 200	360-600	**2-3x penalty**
Branch mispredict	5 - 20	15-60	Pipeline flush
Page fault (minor)	100,000	300,000	Allocate physical page
Page fault (major)	1,000,000+	3,000,000+	Load from disk
TLB miss	10 - 100	30-300	Page table walk

Key insight: Staying in L1/L2 cache is 100x faster than RAM. **Cache locality is everything.**

Java Language Primitives

Operation	Latency (ns)	Notes
Field read (non-volatile)	1 - 2	Direct memory access
Field write (non-volatile)	1 - 2	Store to L1 cache
Volatile read	10 - 20	Memory barrier + cache invalidation
Volatile write	10 - 20	Memory barrier + flush
`synchronized` (uncontended)	25 - 40	Biased locking fast path
`synchronized` (contended)	5,000 - 25,000	OS scheduler involvement
`AtomicLong.get()`	5 - 10	Volatile read
`AtomicLong.set()`	5 - 10	Volatile write
`AtomicLong.incrementAndGet()`	15 - 30	CAS (uncontended)
`AtomicLong.incrementAndGet()`	100 - 500	CAS (4 threads)
`VarHandle.get()`	1 - 2	Plain mode (non-volatile)
`VarHandle.getAcquire()`	5 - 10	Acquire barrier
`VarHandle.setRelease()`	5 - 10	Release barrier

Key insight: Volatile and CAS cost 10-30ns. Locks under contention cost 5-25µs (1000x more!).

System Calls and OS Operations

Operation	Latency (µs)	Notes
`getpid()`	0.5 - 1	Simplest syscall
`gettimeofday()`	0.5 - 1	VDSO-accelerated
`System.nanoTime()`	0.025 - 0.040	25-40ns, clock_gettime
RDTSC read	0.005	5ns, single instruction
Thread park/unpark	5 - 10	OS scheduler
Context switch	5 - 10	Save/restore registers
Thread creation	10 - 50	Allocate stack, OS setup
`mmap()` small	5 - 10	Virtual memory map
`munmap()`	1 - 5	Release mapping

Key insight: Any syscall costs 500ns minimum. Batch them or avoid entirely.

Network Operations (Localhost)

Operation	Latency (µs)	Notes
TCP loopback (small packet)	10 - 20	Kernel network stack
UDP loopback	5 - 10	No connection overhead
Unix domain socket	2 - 5	Bypass network stack
Shared memory (shmem)	0.5 - 1	Fastest IPC

Key insight: For local IPC, shared memory is 10-20x faster than TCP.

Storage Operations

Operation	Latency (µs)	Notes
SSD random read	150 - 300	Flash latency
SSD sequential read	50 - 100	Firmware optimization
NVMe SSD read	10 - 50	PCIe interface
RAM disk read	1 - 5	tmpfs
HDD random read	10,000 - 15,000	10-15ms seek time

Key insight: Even NVMe is 100x slower than RAM. Keep hot data in memory.

Your Ring Buffer Operations (for comparison)

Operation	Latency (ns)	Speedup vs Java Queue
SPSC claim + commit	10 - 15	20-50x faster
SPSC poll	8 - 12	20-50x faster
MPSC claim + commit (4 producers)	30 - 60	10-20x faster
MPSC poll	10 - 15	20-50x faster
LinkedBlockingQueue offer+poll	250 - 500	Baseline (locks!)

This is why you're building custom ring buffers.

1.9 HdrHistogram: Seeing the Whole Distribution

Standard percentiles (`Collections.sort()` + indexing) are fine for small datasets, but they **fail catastrophically** for high-resolution latency tracking.

The Problem with Standard Percentiles

Naive percentile calculation:

```
List<Long> latencies = new ArrayList<>(1_000_000);
// ... collect samples ...

Collections.sort(latencies);
long p99 = latencies.get((int)(latencies.size() * 0.99));
```

Problems:

1. **Memory**: Storing 1 million longs = 8MB per metric
2. **CPU**: Sorting takes O(n log n) = 20 million comparisons
3. **Lock contention**: Synchronizing access to shared list
4. **Coordinated omission**: Missed samples during slow periods

HdrHistogram solves all of this.

What Is HdrHistogram?

HdrHistogram (High Dynamic Range Histogram) is a **constant-space, constant-time** data structure for recording latency distributions with:

- **Fixed memory**: ~3KB for 99.99th percentile at nanosecond precision
- **O(1) recording**: ~10ns per sample
- **No coordinated omission**: Built-in correction
- **Full distribution**: Not just percentiles, entire histogram

Example:

```
import org.HdrHistogram.Histogram;

// Create histogram: max 1 hour (3.6T ns), 3 sig figs
Histogram histogram = new Histogram(3600_000_000_000L, 3);

// Record samples (10ns each)
for (long latency : measurements) {
    histogram.recordValue(latency);
}

// Get percentiles
long p50   = histogram.getValueAtPercentile(50.0);
long p99   = histogram.getValueAtPercentile(99.0);
long p999  = histogram.getValueAtPercentile(99.9);
long p9999 = histogram.getValueAtPercentile(99.99);

// Print distribution
histogram.outputPercentileDistribution(System.out, 1.0);
```

Output:

```
Value   Percentile   TotalCount   1/(1-Percentile)
100     0.000000     1            1.00
150     0.500000     500000       2.00
200     0.900000     900000       10.00
```

```
450     0.990000     990000       100.00
3200    0.999000     999000       1000.00
85000   0.999900     999900       10000.00
```

This shows the **entire distribution**, not just a few percentiles.

Why HdrHistogram for HFT

1. Constant memory (critical for production)

```
Histogram histogram = new Histogram(1_000_000_000L, 3);  // ~3KB
```

You can embed this in every hot path without worrying about memory.

2. Coordinated omission correction

```
histogram.recordValueWithExpectedInterval(
    measuredLatency,
    expectedInterval  // E.g., 10ms if you expect 100 ops/sec
);
```

Automatically compensates for missed samples during pauses.

3. Mergeable histograms

```
Histogram global = new Histogram(1_000_000_000L, 3);
Histogram thread1 = new Histogram(1_000_000_000L, 3);
Histogram thread2 = new Histogram(1_000_000_000L, 3);

// Threads record locally (no contention)
thread1.recordValue(latency1);
thread2.recordValue(latency2);

// Merge periodically
global.add(thread1);
global.add(thread2);
```

No lock contention on the hot path!

4. Percentile accuracy

Standard percentiles on 1M samples give you **~1000-sample resolution** at p99.9 (0.1% = 1000 samples).

HdrHistogram gives you **configurable precision** (3 significant figures = 0.1% error at all percentiles).

Reading HdrHistogram Output

Key columns:

- **Value**: Latency in nanoseconds (or your unit)
- **Percentile**: Cumulative percentage (0.0 to 1.0)
- **TotalCount**: Number of samples ≤ this value
- **1/(1-Percentile)**: How "rare" this percentile is

Example:

```
Value   Percentile   TotalCount   1/(1-Percentile)
450     0.990000     990000       100.00
```

Interpretation:

- 99% of samples took ≤450ns
- 1% (1 in 100) took >450ns
- Out of 1,000,000 samples, 990,000 were ≤450ns

Using HdrHistogram in Your Code

Basic setup:

```
import org.HdrHistogram.Histogram;
import java.util.concurrent.TimeUnit;

public class LatencyTracker {
    private final Histogram histogram;

    public LatencyTracker() {
        // Max value: 1 hour, precision: 3 sig figs
        histogram = new Histogram(
            TimeUnit.HOURS.toNanos(1),
            3
        );
    }

    public void recordLatency(long nanos) {
        histogram.recordValue(nanos);
    }

    public void printReport() {
        System.out.println("Latency Distribution:");
        histogram.outputPercentileDistribution(
            System.out,
            1.0  // Output in nanoseconds
        );
    }

    public long getP99() {
        return histogram.getValueAtPercentile(99.0);
    }
}
```

With coordinated omission correction:

```
public class OrderSubmissionTracker {
    private final Histogram histogram;
    private static final long EXPECTED_INTERVAL_NS = 10_000_000; // 10ms
```

```
    public void recordSubmission(long latencyNanos) {
        histogram.recordValueWithExpectedInterval(
            latencyNanos,
            EXPECTED_INTERVAL_NS
        );
    }
}
```

Thread-local histograms (no contention):

```
public class MultiThreadedTracker {
    private final ThreadLocal<Histogram> localHistogram =
        ThreadLocal.withInitial(() -> new Histogram(3600_000_000_000L, 3));

    private final Histogram globalHistogram =
        new Histogram(3600_000_000_000L, 3);

    public void recordLatency(long nanos) {
        localHistogram.get().recordValue(nanos);
    }

    public synchronized void merge() {
        // Periodically merge thread-local histograms
        localHistogram.get().copyInto(globalHistogram);
        localHistogram.get().reset();
    }
}
```

Chapter 1 Summary: Key Takeaways

You've completed the foundation of performance measurement. Here's what you now know:

Core Concepts

1. **Time scales matter**: 1ns is to 1s as 1s is to 31.7 years
    - Nanoseconds: CPU and cache operations
    - Microseconds: RAM, OS, and local I/O
    - Milliseconds: Network and disk
2. **RDTSC is fastest**: 5ns overhead vs 25-40ns for `System.nanoTime()`
    - Requires calibration (cycles → nanoseconds)
    - Full implementation coming in Chapter 8
3. **Percentiles > averages**: The average is a lie
    - p50 (median): Typical case
    - p99: SLA threshold
    - p99.9, p99.99: Business-critical outliers
4. **Multimodal distributions**: Latency often has multiple modes

    - Cache-hot vs cache-cold paths
    - GC pauses create separate distribution
    - Look for large gaps between percentiles
5. **Coordinated omission**: Measure at a fixed rate or compensate for missed samples
    - Don't stop measuring when the system is slow
    - Use HdrHistogram's built-in correction
6. **Latency vs throughput**: You can't optimize both simultaneously
    - HFT optimizes latency on the critical path
    - Batching increases throughput but adds latency
7. **JMH is essential**: Simple loops are wrong
    - Warmup for JIT compilation
    - Blackhole to prevent dead code elimination
    - Sample mode for percentiles

Tools in Your Toolkit

- **JMH**: Micro-benchmarking framework
- **HdrHistogram**: High-resolution latency distribution
- **perf**: Linux performance counters (preview for Chapter 8)
- **VTune/uProf**: Advanced profiling (coming in later chapters)

Cost Reference Table (Memorize This!)

Operation	Latency	When It Matters
L1 cache hit	1 ns	Always
L2 cache hit	10 ns	Hot paths
L3 cache hit	50 ns	Shared data
RAM (local)	100 ns	All memory access
RAM (remote NUMA)	200 ns	**Multi-socket systems**
CAS (uncontended)	20 ns	Lock-free code
Volatile read/write	15 ns	Visibility guarantees
synchronized (uncontended)	30 ns	Locks
synchronized (contended)	10,000 ns	**Avoid!**
Context switch	5,000 ns	Thread scheduling

What's Next

You now have the measurement framework. **Chapter 2** will teach you modern CPU architecture—why NUMA matters, what cache lines are, and why false sharing kills performance.

You'll understand **why** that 100ns operation sometimes takes 2,000ns (cache miss). You'll learn **why** NUMA remote access is 2x slower. You'll see **why** thread affinity can save you 5-10 microseconds.

With CPU topology knowledge from Chapter 2, you'll then be ready to understand the performance characteristics of different data structures and code patterns—why `ArrayList` is faster than `LinkedList`, why Big O constants matter, and how cache locality dominates algorithm choice in HFT systems.

Welcome to the world of nanosecond-precision Java. Let's build your foundation.

End of Chapter 1

Next: Chapter 2 - Modern CPU Architecture Deep Dive

End of Sample

This sample contains the Preface and Chapter 1.

The complete book continues with eleven more chapters:

CPU fundamentals, cache hierarchy, NUMA architecture, SIMD, thread affinity, topology discovery, and NUMA-aware programming.

Low Latency Programming in Java · Book 1

Amardeep Mond · faster-hft.dev