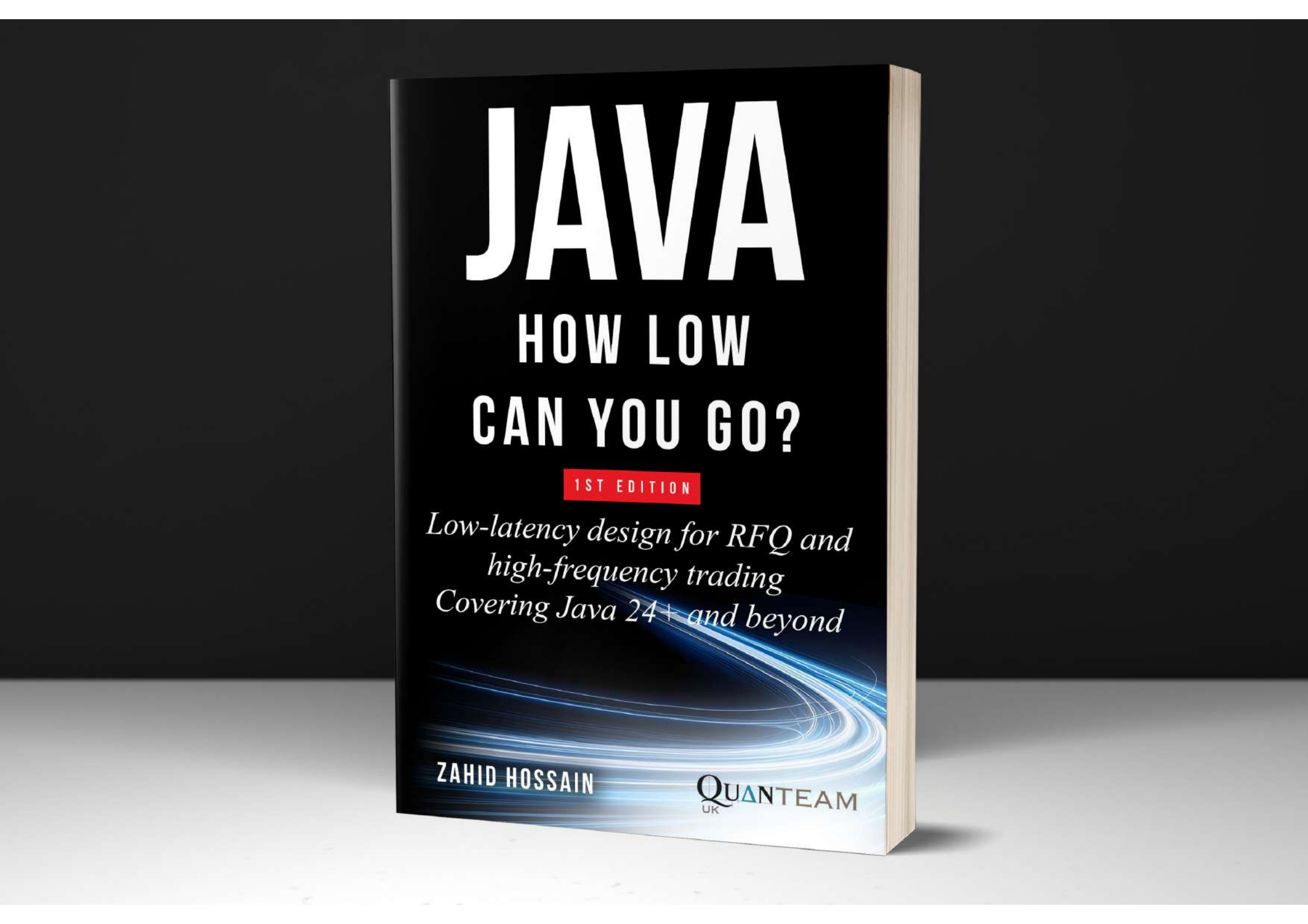

JAVA
HOW LOW
CAN YOU GO?
1ST EDITION
Low-latency design for RFQ and
high-frequency trading
Covering Java 24+ and beyond
ZAHID HOSSAIN
QUANTEAM
UK

1ST EDITION

Zahid HOSSAIN is an entrepreneur, software engineer, and technology leader with over 18 years of experience in e-trading, quantitative development, and building low-latency trading systems at Charles River / State Street, Jefferies, Citi, Credit Suisse, Barclays Investment Bank, and Bloomberg, in London and New York. He has a strong educational background, holding both a Master's and a Bachelor's degree in Computer and Electrical Engineering where he graduated top of his class.
https://www.linkedin.com/in/zahid-hossain/

What is this book about:

This book is a practical guide to ultra-low-latency Java engineering for high-frequency trading, RFQ systems, and market-making desks in banks and hedge funds. It focuses on real techniques used across the industry—from JVM tuning and off-heap design to Unix and network optimization—showing how modern trading firms are moving from C++ to high-performance Java. Rather than theory, it delivers hands-on patterns, architectures, and system-level practices for building deterministic, production-grade latency-critical systems.

Who is this book for

This book is for anyone—from hands-on developers to CTOs. Developers preparing for low-latency Java interviews will find extensive real questions and practical guidance, while senior leaders can focus on the architectural insights without diving into low-level implementation details.

QUANTEAM UK

JAVA: HOW LOW CAN YOU GO?

ZAHID HOSSAIN

JAVA HOW LOW CAN YOU GO?

1ST EDITION

Low-latency design for RFQ and high-frequency trading Covering Java 24+ and beyond

ZAHID HOSSAIN

QUANTEAM UK

Java: How low can you go

Low-latency design for RFQ and high-frequency trading – covering Java 24+ and beyond.

Zahid HOSSAIN

QUANTEAM UK

Java: How low can you go.

First published in 2025.

*This book is dedicated to **Quanteam UK**, whose inspiration and sponsorship made it possible. Quanteam UK is a consultancy firm specializing in the Financial Markets industry.*

— Zahid HOSSAIN

Contributors

About the author

Zahid HOSSAIN is an entrepreneur, software engineer, and technology leader with over 18 years of experience in e-trading, quantitative development, and building low-latency trading systems at Charles River / State Street, Jefferies, Citi, Credit Suisse, Barclays Investment Bank, and Bloomberg, in London and New York. He has a strong educational background, holding both a Master's and a Bachelor's degree in Computer and Electrical Engineering where he graduated top of his class.

Throughout his career, he has been recognized multiple times as an exceptional performer for his contributions to high-impact technology initiatives. Currently based in London, Zahid specializes in low-latency Java development, and he has a passion for algorithms, data structures, financial technology, and AI.

About the reviewers

Antoine DUCHER, founder and CEO of Quanteam UK, a London-based consultancy dedicated to financial markets and quantitative modelling. Antoine brings over 20 years of experience in capital markets and consulting.

Eugenie STEELE, Head of Marketing at Quanteam UK. Eugenie has over 18 years of experience in marketing, having previously worked for BNP Paribas and RBC BlueBay Asset Management, both in France and the UK.

Sabahudin IRFAN, Consultant at Quanteam UK. Sabahudin has many years of experience in financial services and technology, having worked with major institutions such as Barclays, State Street, and others.

Table of Contents

Preface 12

Summary 12

Interview Questions 13

References 14

Java Types 16

Primitive Types 16

boolean 18

Wrapper Types in Java 24

JVM Optimizations in Wrappers 27

JVM caching Mechanism for wrapper types 29

String Type 33

JVM Optimization Around String 35

Streamlined Deduplication Process: 37

Record 41

Some strategies for achieving low latency. 42

Type Checking and Arrays in Low Latency 43

Conclusion 50

Java Collections and Their Underlying Mechanics 51

HashMap 52

JVM HashMap Optimization 54

Resizing Issue 55

ConcurrentHashMap 57

ConcurrentHashMap Synchronization Tricks 58

HashMap Vs ConcurrentHashMap Resizing 61

Fail-Fast vs. Fail-Safe Iteration 62

Internal Mechanics of an ArrayList 64

Low Latency Application's Hot Path 66

CopyOnWriteArrayList 67

Here are some Tricky Interview Questions on collections 70

Threading 74

JVM Thread Deep Dive 74

ForkJoinPool vs ThreadPoolExecutor 77

Virtual Threads 79

ForkJoinPool in JDK 88

Thread Pinning In JVM/Java (CPU Affinity Explained) 91

Thread Priority 93

Some Interview Questions 93

Synchronization 98

How do they work 99

Biased Locking 105

Reader Writer Lock 107

Compare-And-Swap (CAS) 110

CAS vs ReentrantLock Performance 113

Low Latency Synchronization Test 118

Some Tricky Interview Questions 120

The Evolution of Garbage Collection 122

The First People 122

The Reason for G1GC 123
Shenandoah Garbage Collection: The Brooks Forwarding Pointer and Regioned Heap 126
Z Garbage Collection: Colored Pointers and Concurrent Relocation 131
Azul's C4 Garbage Collection: Hardware Concurrency 137
Typical Pause Times of Major Java Garbage Collectors 141
Epsilon GC 143
No-GC Off-Heap Storage 145
Eliminate All Transient Allocations 145
Utilize Off-Heap Memory or Direct Buffers 146
Pre-allocate Object 147
Java's built-in collections (HashMap, ArrayList, etc.) 147
Reuse Charset Encoders and StringBuilders 148
Use Pooled Network Buffers 148
Avoid Lambdas and Anonymous Classes in Hot Code 148
Avoid String Creation and Exception Throwing 149
GC free, Lock free Ring Buffer 150
Conclusion 152
Java Recent Projects 154
Valhalla 154
Leydon 154
Babylon 155
Panama 155
Loom 155
Project Leydon: The JVM's Speculative Nature 156
Project Babylon: Code Reflection and the GPU Future of Java 158
Project Amber and Low latency Application 162
Azul's Zing in HFT 163
Trading Systems 166
RFQ System Architecture 169
Market Making Architecture 172
How Exchanges match orders 175
Darkpool 179
Smart Order Router 187
Algorithmic Execution Strategies 191
VWAP - Volume Weighted Average Price 193
TWAP - Time Weighted Average Price 195
POV Participation of Volume 196
Client Order Lifecycle - Combining the Parts 198
Ultra Low Latency Code 202
What is ultra-low latency 202
Low latency business in finance 203
Programming Language 204
Challenges of Java and other consideration 205
Low Latency libraries in Java 206
Zero Copy 208

CPU and Memory consideration 209
Method Inlining 216
Loop unrolling 218
Escape analysis 220
Branch Prediction 220
SIMD operation (Java 24+) 223
Thread Pinning (CPU Affinity) 226
General Optimizations for Low-Latency Systems 228
OS (LINUX) Tuning 232
The importance of OS tuning 232
The concept of Jitter 233
Measure Before You Tune 234
Red Hat Linux – Tuned Profiles 234
NUMA Awareness 236
BIOS Configuration 239
CPU Isolation 242
Cage your GC Thread 246
Avoiding Interference (De-Jittering) 247
Huge Pages — Making Memory Access Faster 248
Kernel Timer Tick (nohz_full) 249
Conclusion 252
Market Data & Pricing Engine 254
Market Data – UDP vs TCP 255
Market Data Format 257
Architecture of Market Data – Pricing Engine 258
Conclusion 268
Network Tuning 270
Typical Packet Flow 271
Kernel Overhead 273
Kernel Bypass 275
Interrupt mode vs Poll Mode 275
Smart NIC cards 277
Linux Tuning 278
Conclusion 281
KDB+ A Low-Latency Database 283
High-frequency Trading 284
KDB+ Architecture 285
KDB+ processes 288
Why KDB+ is fast 288
How Compression Works in kdb+ 290
Understanding the Q Language 292
Conclusion 294
INDEX 296

This page intentionally left blank

PREFACE

SUMMARY

This book reflects many years of building and tuning low-latency trading systems on the premises of some of the most prominent investment banks in the world. It includes firsthand engineering decisions and production environments where every microsecond counts. For most of my professional life, I have straddled the domains of Java performance, electronic trading, and system architecture. This book, or rather the subsequent chapters, sets out to teach the readers the advanced features of Java types, memory layouts, JVM optimizations and how the types like wrappers, primitives, records and strings work internally along with insight into JIT compiler and the memory model. It uses practical case studies to demonstrate the principles of bounded garbage, predictable performance in latency sensitive environments, reliable, and cache aware.

The subsequent chapter delves into the core libraries in addition to the concepts of threading and synchronization, detailing the operational mechanics of collections, the processes during resizing, and the significance of fail-fast and fail-safe practices in overload conditions. Then, the book describes the use of concurrency mechanisms in Java from biased locking and CAS to modern virtual threads along with CPUs pinned to threads. These themes are augmented by the history of garbage collection from G1GC to Shenandoah, ZGC, and Azul's C4, along with techniques to remove GC pauses through off-heap and zero-copy methods. Every chapter provides an equilibrium of theory

and low-level practical tuning, supported by reasoning from benchmarks.

The final chapters unify the theory underpinning Java performance with actual trading infrastructure followed by OS and network tuning. It describes the processes of order matching in addition to the architecture of RFQ and market-making systems, the execution of market algorithms such as VWAP and TWAP on low-latency stacks, and the structuring of algorithmic strategies, Smart Order Router (SOR) etc. It has demonstrated consistent performance through OS-level tuning, NUMA, CPU isolation, and network optimization with Kernel bypass. The book concludes with an outline of Java code optimization and an integrated perspective on JVM systems competing in high-frequency and algorithmic trading where every microsecond is pivotal to profit-making.

Interview Questions

This volume contains more than just system design, it also contains hundreds of real interview questions alongside intellectual exercises - not fictitious enigmas, rather questions I have asked, and I have been asked, during my decades of employment mentoring and hiring global engineering trading teams. Engineers trying the challenging field of low-latency systems will find these questions valuable, as will interviewers seeking to assess a candidate's real-world capability. Each question touches on a real-life design problem, system performance obstacle, or JVM instance that is observable in production. This book tries to equip the readers with practical insights that only come with years spent in highly demanding, low-latency, and high-pressure work situations in addition to technical knowledge.

References

While writing this book, I undertook some additional research on the topics, including watching several YouTube lectures. I found that some of these sources had already conducted similar types of benchmark tests that I aimed to conduct. To avoid unnecessary duplication of work, I have appropriately integrated those benchmark findings. I have transformed some diagrams in the book based on similar ideas found in those lectures, although I have changed them to better reflect my explanations by adding or removing elements. I would like to thank these creators for their valuable contributions to the low latency community. Lastly, the title of this book was inspired by one of the lectures that particularly resonated with me. For those who wish to pursue these materials, I have collected the list of videos below.

Understanding the Disruptor
https://www.youtube.com/watch?v=DCdGlxBbKU4

Kernel-bypass techniques for high-speed
https://www.youtube.com/watch?v=MpjlWt7fvrw

Ultra-low latency Java in the real world - Daniel Shaya
https://www.youtube.com/watch?v=BD9cRbxWQx8

Low Latency Market Data
https://www.youtube.com/watch?v=y-BSb045LNk

Network Performance Tuning
https://www.youtube.com/watch?v=ZYCKSN4xf84

This page intentionally left blank

JAVA TYPES

The most central part of the Java type system is its small set of primitive types, which form the foundation of all data structures. In Java, any variable that is declared must first be given a type, which is a form of pact. This pact defines the total amount of memory allocated, the maximum and minimum values that can be stored, and the functions that can be performed on the variable.

Since Java was developed for use on any type of hardware, the size of each primitive type is bound by the language specification itself. This is determined by the Java Virtual Machine, not by the processor or the operating system. A 32-bit int is thus, and for example, exactly 32 bits on a desktop Java Virtual Machine (JVM) and on an embedded controller. This was one Java's main selling points back in the day, the concept of write once, run anywhere.

PRIMITIVE TYPES

Primitive types are not objects and exist in system memory as opposed to the heap. They do not possess method tables, identity, headers, or any other form of metadata. Primitive types are composed of pure values, which is the reason as to why they are extremely fast and compact. Advanced Java programmers are thus able to perform any numeric operation and use loops or even counters without any form of reduced performance by using primitive types instead of wrapper classes.

NO-GC OFF-HEAP STORAGE

In any ultra-low-latency system such as, HFT, RFQ or continuous market making, every microsecond counts. For general-purpose Java applications, ‘garbage collection’ (GC) involves unpredictable pauses. This is unacceptable on the hot path of any trading engine. Whether the JVM is using ZGC, Shenandoah, or Azul's C4 collector, these still incur safe point coordination and metadata barriers. In an HFT engine, a 50-microsecond pause is already catastrophic. This results in missed quotes, stalled prices, ignored order delays, or out-of-order response acknowledgments.

Hence the term "Zero-GC Java" is used not to describe a globally disabled GC function, but rather, critical path design in which the Java Virtual Machine (JVM) has ‘nothing’ to collect. These sections of code are usually the order routing loops, engine matching, and market data ingestion. In these loops, all transient object creation must be erased. Reliance on object reuse, off-heap data structures, and pre-allocation is what is needed.

Below are the main techniques to implement ‘true’ Zero-GC behavior on a Java hot path.

ELIMINATE ALL TRANSIENT ALLOCATIONS

To never have to invoke garbage collection, never allocate. Slow loops with hot vertices suffer from heap churn due to every new operation. You should not allocate like those below:

JAVA RECENT PROJECTS

Shifting Focus of Java Projects within Recent Years.

Java is a language which has been greatly improved in the last few years. The development of OpenJDK has systematically improved Java by targeted ‘sub’ projects which aim to improve the underlying JVM model. The modernization of Java OpenJDK projects aims to shift the use of Java to lower latency in a compute cloud, while enhancing native use and concurrency with improved JVM CPU profiling. The projects modernize Java to achieve higher efficiencies and performance compute Java workloads.

Compute intensive workloads such as trading systems and numerical analytics can massively benefit from the use of projects. I will begin with a basic introduction, followed by a deep dive into their synergies in low-latency environments.

VALHALLA

Java 24 and above implements **value-based classes** which allow for objects without identity (ex. class instances missing object identity). The use of Valhalla’s classes greatly improves **memory cache** and reduce the overhead placed on the heap while also improving the **locality** of the memory cache.

LEYDON

Project Leyden (Java 24+) addresses JVM's slower startup and warm-up delays. The approaches of static image generation and AOT initialization help reduce the time that Java applications take to reach

TRADING SYSTEMS

Every microsecond counts towards competitiveness in high-frequency trading (HFT). The order to market systems are designed to receive and process market data, make trading decisions, and final orders to the exchange in as little as 10 microseconds- a complete process. Understanding exchange microstructure, intricate matching algorithms, and advanced systems design techniques (kernel-bypass networking, CPU pinning, cache-friendly data layouts, and lock-free, GC-free data structures) are fundamental for achieving this performance. HFT infrastructure usually consists of collocated servers positioned in the data center of the exchanges to minimize network latency and ensure deterministic execution. There is a strong focus on speed, with stable latency in the single-digit microsecond range because high-frequency trading is extremely sensitive to delays. Even tiny spikes in latency can cause missed trades or outdated quotes.

Unlike trade execution systems, counterpart and order management systems like Charles River, Bloomberg AIM, ION and Fidessa operate under different latency thresholds and focus on the millisecond range, rather than the microsecond range. Luckily, I had the privilege to work on all of them throughout my career. These systems care about speed only to the extent that performance still meets requirements for reliability, auditing, and regulatory compliance. These systems also address the management of large order streams, portfolio allocations, and multi-exchange and dark pool connectivity. Their primary focus is on execution quality and expense, rather than timing precision. These

operators to submit completed trades for verification for post-trade transparency in real time. Operators must also conduct and enforce strict, non-discriminatory, access control to order information, implement malicious order information systems, and ensure no client discrimination access policies.

However, dark pools have become critical in the new configurations of the micro-structure of many modern exchanges. It also needs to be understood that dark pools are neither illegal nor expressly unfair, they are simply used differently. If public repositories are compared to Times Square and its noise and activity, dark pools are private art galleries and meeting rooms that facilitate the completion of serious deals hidden from the public eye. Each of these places is critical in their own form. One gives the price; the other provides the volume.

For many traders, dark pools have become another reminder that the publicized market is only a small percentage of the reality. Not every volume of trade is visible nor every volume surfaced during the price discovery process is public, public transactions where the real market is neither dark nor light. Therein lies the true market structure of contemporary trading. To optimally design a low latency trading system, both market conditions must be understood.

Major Dark Pools in International Markets

There are differences between dark pools, some are owned by global investment banks while others are owned by independent technology companies or exchanges. Each has its own clientele, regulatory coverage and matching logic. Below are some dark pools that have had an impact on institutional trading over the last ten years.

Credit Suisse **Crossfinder**

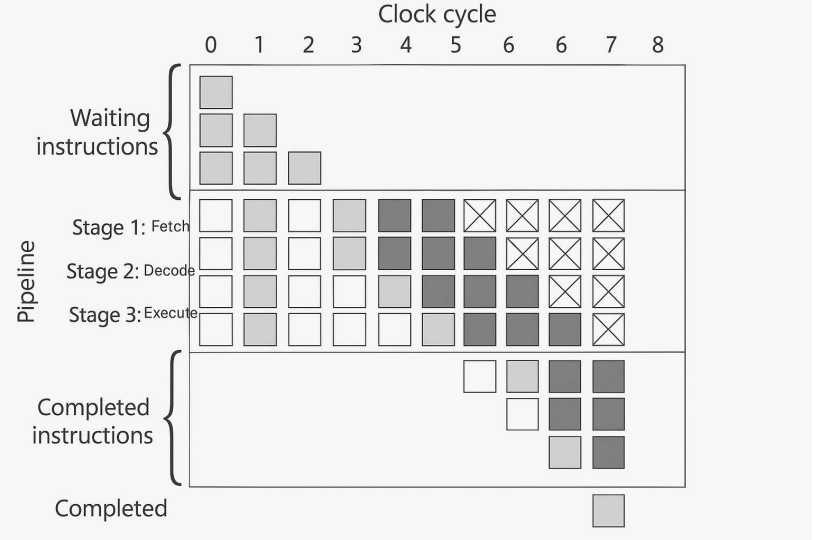
ULTRA LOW LATENCY CODE

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." - Donald Knuth

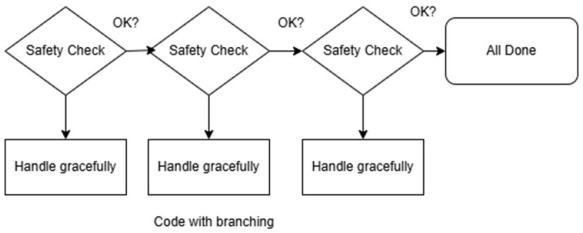
Caring about that 3% is not premature optimization in my sense; it is the early realization that low latency begins with awareness. Recognizing small inefficiencies before they accumulate is what separates ordinary systems from truly optimized ones.

WHAT IS ULTRA-LOW LATENCY

Within the domain of 'Ultra-low Latency Programming', software performance is sub-100 microseconds and every microsecond is of utmost importance. Response times are not only important on average; there are tail latencies to be dealt with, even the 1 out of 10,000 events that take a tad longer to process. Such systems, high-frequency trading engines and real-time risk processors, are not only built for speed; but also, for the predictably consistent performance termed the four nines (99.99%) of latency stability. In high precision scenarios, questions deepen, for instance, whether Java is a suitable language for ultra-low latency systems, what programming paradigms are necessary to achieve such a level of determinism, and do microservices or a distributed architecture help or hinder these objectives. Such questions define the ultra-precision art and engineering necessary to build systems with failures and successes separated only by a matter of nanoseconds.



I consider Ultra-low latency is a programming design pattern regardless of programming language. For example, branch-less coding, lock-free coding, no GC coding etc, they are all science rather than engineering. Consider the following picture with lots of branches:



Code with branching

Can you make this better?

OS (LINUX) TUNING

As you strive towards achieving microsecond-level determinism, variability becomes your enemy. Linux is an outstanding general-purpose operating system. Yet, under its default settings, it predominantly optimizes for fairness relative to resource allocation, throughput, power efficiency, and not, latency predictability. In ultra-low latency trading, we have to convert Linux from being multi-tenant, time-sharing to being deterministic, single-tenant.

In this chapter, we understand how to transform Java trading engines on a standard Linux installation into a more predictable environment. The focus is on incoherent microsecond precision, wherein pinned threads, aligned NUMA, configured IRQ, and tuned BIOS collectively erased the variability towards achieving consistent micron level performance.

THE IMPORTANCE OF OS TUNING

For the best Java code, fastest network card, and lowest-latency queue to matter, the operating system must not interrupt at the worst time. 100–500 µs delay, incurred by a background kernel thread, an errant interrupt, or some fuel-saving power throttling, is an eternity in trading.

OS tuning is not simply about making Linux "faster." It is about making it predictable. We do not stand for "average fast." We want always fast. Enhancing a system's performance doesn't always require code

MARKET DATA – UDP VS TCP

TCP vs. UDP – Reliability vs. Speed: When transporting market data across a network, TCP and UDP are the two predominant options available.

Designed primarily for accuracy, TCP (Transmission Control Protocol) is responsible for tracking every individual packet utilizing sequence numbers, and issues an acknowledgment (ACK) for each individual packet. In the case something is lost or arrives out of sequence, TCP will also retransmit the packet. This is what makes TCP very reliable. However, the system's overall latency becomes higher and more unpredictable because each packet necessitates back-and-forth communication. Because of these characteristics, TCP is suited for scenarios such as file downloads and web traffic, but not for environments where every microsecond is highly critical.

UDP (User Datagram Protocol), on the other hand, entails no such retransmission, acknowledgment, or sequencing. Streamlined systems can simply send packets for an undetermined amount of time without check. This is also called “fire and forget”. This behaviour is what makes UDP very appealing in high-frequency trading (HFT) and the overall market. In systems where speed is more critical than reliability, UDP is preferred. This is because, without any check-in system, UDP loses packets on the system with no possible retransmission. This means that the application must recognize the lost packets and have the protocol to recover it.

How Exchanges Send Market Data

To send market data, exchanges employ UDP multicast. With this technology, a single data packet can be sent to hundreds and even thousands of recipients simultaneously. Each market data update - be

Network Tuning

One of the most critical steps in high-frequency trading system development is the optimization of the network since it is the primary and terminal component of each trade execution cycle. We're talking about systems that handles even more than millions of packets per second, so every microsecond of overhead matters. In high-frequency trading, market data is received and processed in real-time, and orders are transmitted over the same network. Consequently, delays, jitter, and loss of data packets all negatively impact the profit or even profit can turn into loss. The optimized performance of the JVM, code and OS tuning do not mitigate the adverse effects of a slow or unstable network. Additional latency spikes are a result of added interrupts, context switching, and overheads from the TCP stack, all of which can take microseconds.

During this chapter, we examine techniques that are kernel-bypass, which are strategies that allow user-space code to interact directly with a network card without utilizing any of the operating system's networking stack. Kernel-bypass is not a new concept; it has been implemented over the past decade in leading trading companies. As engine designer for low latency systems at Credit Suisse more than a decade ago during 2011, it was my objective to eliminate the kernel path and allow the application to communicate directly with the network hardware. Kernel-bypass provides the ability to eliminate interrupts, system calls, and data copies, which facilitates the faster arrival of packets into user memory for direct processing by a polling thread. There are hardware solutions, such as the Solarflare (currently Xilinx) network interface cards (NICs), also called smartNIC, together

KDB+ A Low-Latency Database

When it comes to the sheer volume of data generated by today's systems, traditional databases just don't cut it. Financial markets, industrial sensors, telecom networks and online platforms all fire off data in a continuous flow of time-stamped events, and the challenge isn't so much about storing that data, but about crunching it down to earth in real-time. Well-known decision-making environments that require millisecond. Sometimes even microsecond, responses mean that the database is basically on par with the application logic sitting above it.

Kdb+ was built from the ground up to tackle this sort of data and is often referred to as a high-performance **time-series database**, but that doesn't tell the full story. It's the architectural ideas that enable kdb+ to gobble, slice and dice and send back out enormous amounts of real-time data with hardly any delay that set it apart. Coming hotfooting into the world of finance, kdb+ is at its most visible, but it's being used in manufacturing, aerospace, telecommunications networks and lots of other places where the cost of delay is a serious issue.

Two things drive kdb+'s lightning-fast performance: it's purely in-memory, with the most recent data sitting snuggly in RAM and being able to be queried in less than a millisecond, and it’s column-based rather than row-based. Since time-series workloads tend to involve

```
select ma5: 5 mavg bid by sym from quote
```

Output may look like this:

sym	ma5
AAPL	150.10
AAPL	150.125
AAPL	150.15
AAPL	150.1425
AAPL	150.15
AAPL	150.18
MSFT	305.50
MSFT	305.525
MSFT	305.55

VWAP (Volume-Weighted Average Price): Consider a **Trade** table

time timestamp
sym symbol
price float
size int

the basic VWAP for each symbol is:

```
select vwap: sum price * size % sum size by sym from trade
```

Output may look like this:

sym	vwap
AAPL	150.146
MSFT	305.553

A time-bucketed VWAP (for example, 1-minute intervals) is just as straightforward:

```
select vwap: sum price * size % sum size by sym, 1 xbar time.minute from trade
```

The expression 1 xbar groups trades into 1-minute buckets before calculating VWAP within each group.

Q functions as the query and control language, which manages how data is loaded, transformed, stored and queried, now or in the past, when working with kdb+. Coming running over off the heels of the data load, q's ability to run directly on columnar memory arrays and shun unnecessary overheads gives developers the chance to write lightning-fast analytics code with the absolute minimum amount of code.

The fusion of language, engine and storage model in kdb+ is the secret to its success in providing extremely low-latency, high-volume time-series analytics.

Conclusion

Speed is not just an advantage, it's basically a necessity, when working in the high-stakes world of high-frequency trading and electronic markets. Financial firms count on databases that can seamlessly keep up with the torrent of live market data, crank out microsecond-level analytics, and deliver reliable results even under the most extreme loads.

Well-known as the de-facto standard in major banks and trading companies, kdb+ has become the go-to for all these reasons. Its in-memory, column-structured layout, merged with the expressiveness of the q language, allows teams to cut through the noise of market ticks, compute rolling statistics and dissect risk in real time without hitting the brakes on the trading engine. Today, numerous global banks, market makers, and quant desks use kdb+ as the anchor of their analytics stacks.

In my time at Citi working in Equity Derivatives, kdb+ was at the heart of real-time monitoring of Greeks, intraday risk and market-sensitive signals and I have found that its low-latency intake, lightning-fast

vectorised queries and efficient storage make it one of the handful of systems that can handle the scale and speed needed in modern low-latency application.

INDEX

:+PrintInlining, 210
+AlwaysPreTouch, 242
Aeron, 143, 145, 163, 179, 181, 192, 194, 200, 201, 202
Affinity, 90
Agrona, 65, 84, 143, 145, 163, 180, 200, 201, 202
Algorithmic Execution Strategies, 186
Anonymous Classes, 145
ArrayList, 64
Autoboxing, 29
Babylon, 151
Biased Locking, 104
BigDecimal, 24
BIOS, 225, 227, 232, 233
BitSet, 21
boolean, 17, 19, 20, 21, 22, 26, 41, 68, 106, 109, 110, 216
Boolean, 19, 26, 27, 30
boxing, 40, 51, 65, 144, 223
Branch Prediction, 214
C4 Garbage Collection, 134
Cache filling, 207
CAS, 13, 56, 58, 59, 60, 67, 98, 99, 103, 109, 110, 112, 113, 117, 118, 222
Challenges of Java, 199
Chronicle Queue, 181, 200, 202
Citi, 177, 186
Client Order Lifecycle, 192
Compare-And-Swap, 109
Concurrent Mark-Sweep, 121
ConcurrentHashMap, 56

contiguous memory, 44, 46, 204
Continuous Compaction, 136
CopyOnWriteArrayList, 67, 69, 70, 72, 73
CountdownLatch, 114
CPU Affinity, 90, 219
CPU Isolation, 235
Credit Suisse, 177, 186, 263
Crossfinder, 177
C-states, 233
CyclicBarrier, 114, 115
dark pool, 161, 174, 175, 176, 177, 178, 179, 180, 182, 184, 193
Darkpool, 174, 178
Deduplication, 37, 38
De-Jittering, 240
Direct Buffers, 143
Drop Copy, 185, 186, 193, 194
Encoders, 144
Epsilon GC, 140
Escape analysis, 213, 214
Exception Throwing, 146
Fail-Fast, 62
Fail-Safe, 62
Fairness, 102
False sharing, 95
FixTag, 40
ForkJoinPool, 77, 78, 79, 80, 86, 87, 88, 89, 90
Forwarding Pointer, 124, 125
G1GC, 121
Garbage Collection, 120
GC free, 147
GC overhead, 31
Goldman Sachs, 178
HashMap, 33, 52, 53, 54, 55, 56, 60, 61, 62, 63, 71, 72, 106, 144, 222
HashMap Optimization, 54
Hot Path, 66

Huge Pages, 241, 242
IRQ affinity, 219, 228, 273
isolcpus, 235, 236, 237, 244, 245
Java Collections, 25, 51
Jitter, 92, 226, 240
JVM caching, 29
JVM Optimization, 35, 103
JVM Optimizations, 28
JVM Parameters, 36
JVM's Speculative Nature, 152
Kernel Timer Tick, 242
Lambdas, 145
Lead Market Maker, 173
Leydon, 150
LMAX, 147, 148, 163
Load Barriers, 135
Lock Acquisition, 102
Lock flag, 98
Lock free, 147
Lock optimizations, 152
Loom, 151
Loop optimizations, 152
Loop unrolling, 212
Low latency business, 197
Low Latency libraries, 200
Mark word, 23, 24, 98, 127
Market Data, 247
Market Making, 162, 167, 168
match orders, 170
Method inlining, 152
Method Inlining, 210
Midpoint Calculation, 180
modCount, 62
NBBO, 175, 177, 180, 181
No-GC, 142
nohz_full, 245

NUMA Awareness, 229
NUMA node, 229, 230, 232
NUMA server, 232
Off-Heap Memory, 143
OpenHFT, 91
OTC derivatives, 163
Panama, 151
Polluted data, 254
POV, 184, 186, 187, 191, 192, 193, 194, 195
Predictable Performance, 123
Pricing Engine, 247
Primitive Types, 16
Pro Rata, 171
P-states, 233
Question, 21, 22, 23, 24, 29, 31, 32, 33, 34, 38, 46, 47, 48, 49, 55, 57, 60, 65, 82, 83, 86, 88, 93, 94, 95, 97, 100, 102, 106, 108, 113, 114
Reader Write Lock, 106
record, 41, 113, 158, 159, 205
Red Hat Linux, 227, 243
Reentrant Lock, 99
ReentrantLock, 112
Reporting and Compliance, 181
Request for Quote, 162
Resizing Issue, 55
RFQ, 2, 14, 85, 140, 162, 163, 164, 165, 201, 207, 221
Ring Buffer, 147
Shenandoah, 124
SIMD, 36, 66, 71, 151, 157, 160, 199, 213, 217, 218, 219
Smart Order Router, 14, 182, 183, 187, 193, 194
SOR, 183, 184, 187, 189, 190, 193, 194
stamped Lock, 113
String Type, 33
Synchronization, 97
Synchronization Tricks, 58
synchronize, 97

System Architecture, 163
System.arraycopy, 44, 45, 66, 71
TCP, 248
Thread Pinning, 90, 219
Thread Priority, 92
Threading, 74
ThreadPoolExecutor, 78
Time Weighted Average Pric, 189
Tuned Profiles, 227
Turbos Boost, 233
TWAP, 14, 186, 187, 189, 190, 194, 195
Type Checking, 43
UBS ATS, 178
UDP vs TCP, 248
Ultra Low Latency, 196
ultra-low latency, 140, 160, 196, 225, 245, 268, 269, 271
unboxing, 27, 29, 51, 223
UNIX tunning, 220
UnlockExperimentalVMOptions, 141
UseEpsilonGC, 141
Valhalla, 150
Value Class, 39
Virtual Threads, 79
Volume Weighted Average Price, 187
VWAP, 14, 164, 184, 186, 187, 188, 189, 190, 192, 193, 194, 195
Warm-up, 207
Wrapper Class, 26
Wrapper Types, 25
-XX
  +OptimizeStringConcat, 37
  +UseStringDeduplication, 37
  MaxInlineSize, 211
  StringTableSize=<entries>, 37
-XX:+PrintStringTableStatistics, 36
-XX:+UseLargePages, 242
-XX:AutoBoxCacheMax, 30

-XX:MaxInlineSize, 211
Z Garbage Collection, 128
Zero Copy, 201
Zero-GC, 142
Zing, 139, 159, 213