

Contents

Chapter 0: The C++ Abstract Machine	2
Prerequisites for Correctness in High-Performance Code	2
Who Should Read This Chapter	2
Self-Assessment (2 Minutes)	2
Why This Matters for Low-Latency Code	3
0.1 The Abstract Machine Model	3
What This Means	3
Observable Behavior	4
The As-If Rule	4
0.2 Undefined Behavior: The Compiler's License to Assume	4
The Compiler's Reasoning	4
UB Allows Time Travel	5
An Illustrative Bug	5
0.3 Common Undefined Behavior in Performance Code	6
1. Strict Aliasing Violations	6
2. Data Races	7
3. Integer Overflow (Signed)	8
4. Out-of-Bounds Array Access	9
5. Use-After-Free and Dangling References	9
6. Uninitialized Variables	10
0.4 Debug vs Release: The Behavior Gap	11
Why Debug Builds Hide Bugs	11
Illustrative Example	12
Testing Strategy	12
0.5 Optimization and the As-If Rule	13
Loop Optimizations	13
Function Inlining	13
Dead Store Elimination	14
Constant Propagation	14
Loop Invariant Code Motion	14
The Dark Side: When UB Breaks Optimizations	15
0.6 Tools for Detecting Undefined Behavior	15
1. Sanitizers (Runtime Detection)	15
2. Static Analysis	16
3. Compiler Warnings	17
4. Runtime Assertions	17
5. Fuzzing	17
Development Workflow	18
0.7 Case Studies: Common UB Bug Patterns	19
Case Study 1: The Disappearing Null Check	19
Case Study 2: The Overflow That Wasn't Checked	20
Case Study 3: The Data Race in Position Tracking	21
Case Study 4: The Strict Aliasing Bug	22
0.8 The Optimization Paradox	23
The Tension	23
The Solution: Correct First, Fast Second	23
Remember	23
0.9 Compiler Flags for Development and Production	24
Development Flags	24
Production Flags	24
CMake Configuration	25
Summary	25

Key Takeaways	25
Checklist Before Optimizing	26
What's Next	26
Further Reading	26
Essential	26
Deep Dive	26

Chapter 0: The C++ Abstract Machine

Prerequisites for Correctness in High-Performance Code

“Premature optimization is the root of all evil, but so is assuming your code does what you think it does.” — Adapted from Donald Knuth

Who Should Read This Chapter

Read this chapter if: - You’ve encountered “works in debug, crashes in release” bugs - You use `-O3` without fully understanding the implications - You’re not 100% confident about undefined behavior rules - You’ve written multithreaded code without proper atomics - You want to understand why the compiler transforms your code

Skip to Chapter 1 if: - You’ve read the C++ standard sections on undefined behavior - You can explain strict aliasing rules from memory - You regularly use sanitizers and analyze compiler output - You scored 5/5 on the self-assessment below

Self-Assessment (2 Minutes)

Test your knowledge before deciding whether to read this chapter:

Question 1: Is this code undefined behavior?

```
float f = 1.0f;
uint32_t bits = *(uint32_t*)&f;
```

Question 2: Can the compiler remove the null check in release mode?

```
void process(int* ptr) {
    int value = *ptr;
    if (ptr == nullptr) return;
    use(value);
}
```

Question 3: Is this undefined behavior?

```
int x = INT_MAX;
x = x + 1;
```

Question 4: What’s the difference between `volatile` and `std::atomic`?

Question 5: Can the compiler reorder these statements?

```
int data = 0;
std::atomic<bool> ready{false};

data = 42;
ready.store(true, std::memory_order_relaxed);
```

Answers: 1. **Yes** - strict aliasing violation (use `std::bit_cast` or `memcpy`) 2. **Yes** - `*ptr` runs before the check and dereferencing null is UB, so the compiler assumes `ptr != nullptr` and deletes the check. (If the check came first, it would have to stay.) 3. **Yes** - signed integer overflow is undefined behavior 4. **volatile stops the compiler from eliding or merging accesses (for memory-mapped I/O) but gives no atomicity or inter-thread ordering; std::atomic provides both** 5. **Yes** - relaxed ordering allows reordering (use `release` to prevent)

Scoring: - **5/5 correct:** You likely know this material. Skim or skip to Chapter 1. - **3-4 correct:** Read the sections on topics you missed. - **0-2 correct:** Read this chapter carefully before proceeding.

Why This Matters for Low-Latency Code

In typical backend services: - Latency budget: 10-100ms per request - Undefined behavior might cause occasional crashes - Debug vs release differences usually get caught in testing - Performance loss from defensive programming is acceptable - Can afford to be conservative with optimizations

In high-frequency trading: - Latency budget: 1-10 s for entire pipeline (1,000-10,000x faster) - Undefined behavior can cause **silent data corruption** (wrong prices, incorrect positions) - **Cannot reproduce** many bugs under debugger (works at `-O0`, fails at `-O3`) - **Cannot afford** defensive code (every nanosecond counts) - Aggressive optimization is **mandatory**, not optional - Compiler transformations are **essential** to hitting latency targets

The paradox: We need aggressive optimization (`-O3`, `-march=native`, LTO) to achieve sub-microsecond latency, but aggressive optimization exposes undefined behavior mercilessly.

The stakes: In HFT, a single bug can cost millions of dollars in minutes. Code that “seems to work” can fail catastrophically in production.

0.1 The Abstract Machine Model

C++ is not defined by what x86 assembly code executes. It’s defined by an **abstract machine** — a formal model of program execution described in the C++ standard.

What This Means

You write C++ source code:

```
int compute() {  
    int x = 5;  
    x = x + 1;  
    return x;  
}
```

The compiler doesn’t have to generate:

```
mov eax, 5  
add eax, 1  
ret
```

It can generate anything with the same observable behavior:

```
mov eax, 6      ; Computed at compile time!  
ret
```

Or even:

```
; Nothing! Function inlined and constant-folded away
```

Observable Behavior

The abstract machine defines what constitutes “observable behavior”: - **Volatile accesses** (hardware registers, memory-mapped I/O) - **Data written to files** by the time the program terminates - **Interactive I/O** (prompts appear before the program waits for input)

Atomic operations are *not* on this list. Their ordering guarantees constrain what other threads may observe, but the compiler is still allowed to, for example, merge two adjacent relaxed stores to the same atomic.

Everything else: The compiler can reorder, eliminate, transform, or optimize in any way that preserves observable behavior.

The As-If Rule

As-if rule: The compiler can transform your code in any way, **as long as** the observable behavior is preserved, **assuming your program has no undefined behavior.**

Example 1: Constant folding

```
int result = 2 + 3 * 4; // Written by programmer

// Compiler generates:
int result = 14; // Computed at compile time
```

Example 2: Dead code elimination

```
int x = 5; // Written
x = 10; // Overwrites previous value
use(x);

// Compiler generates:
int x = 10; // First assignment removed
use(x);
```

Example 3: Loop optimization

```
int64_t sum = 0; // int would overflow: the result is 499,999,500,000
for (int64_t i = 0; i < 1000000; ++i) {
    sum += i;
}

// Compiler can replace with closed-form formula:
int64_t sum = 1000000LL * 999999 / 2; // Computed at compile time!
```

Implications for HFT: These optimizations are why we can write high-level C++ code and still achieve assembly-level performance. The abstract machine model gives the compiler freedom to generate optimal code.

0.2 Undefined Behavior: The Compiler’s License to Assume

Undefined behavior (UB) is **not** “implementation-defined” or “unspecified.”

Undefined behavior means: You violated the contract. The standard makes **no guarantees** about what happens. The compiler can assume UB never occurs.

The Compiler’s Reasoning

When the compiler encounters a code path that would trigger UB, it can assume that path is never taken:

```

void process(int* ptr) {
    int value = *ptr;    // Dereference first...

    if (ptr == nullptr) {
        return;          // ...then check. Compiler reasoning:
                          // "Dereferencing null is undefined behavior"
                          // "ptr was already dereferenced, so it cannot be null"
                          // "Therefore this check is dead code"
    }
    use(value);
}

// With -O2/-O3, optimized to:
void process(int* ptr) {
    use(*ptr);    // Null check removed!
}

```

Why this happens: The compiler sees the dereference `*ptr` and concludes that `ptr` must be non-null (because dereferencing null is UB). Any check that comes *after* the dereference is therefore redundant. GCC does this under `-fdelete-null-pointer-checks`, which is on by default at `-O2`.

What it does not do: A check that comes *before* the dereference stays. `if (!ptr) return; *ptr = 42;` is correct and keeps its check, because the dereference only runs on the non-null path.

Real-world impact: A refactor that moves a field read above an existing null check silently deletes the check. This is how the Linux kernel's tun driver bug (CVE-2009-1897) became exploitable.

UB Allows Time Travel

UB anywhere on an execution path makes the *whole* path undefined, so the compiler can use it to change code that runs **before** the UB:

```

int table[4];

bool exists_in_table(int v) {
    for (int i = 0; i <= 4; ++i) {    // Off-by-one: reads table[4]
        if (table[i] == v) return true;
    }
    return false;
}

// Compiler reasoning:
// "Reading table[4] is UB, so the loop must return before i reaches 4"
// "The only way to return early is to find v"
// "Therefore the function always returns true"

// GCC -O2 compiles it to:
bool exists_in_table(int v) {
    return true;
}

```

The first four comparisons, which are perfectly valid on their own, are gone too.

An Illustrative Bug

Scenario: Market data parser (a composite of a common bug class, not a specific incident)

Code:

```
// Parse price from wire format (scaled integer)
int64_t parse_price(const char* data) {
    int32_t price_scaled = read_int32(data); // Price in cents
    int32_t multiplier = 100;

    int64_t price = price_scaled * multiplier; // BUG!
    // Multiplication happens in int32_t, then converted to int64_t
    // If price_scaled > 21,474,836, this overflows (UB!)

    return price;
}
```

Symptom: Occasional wrong prices (negative or wildly incorrect)

Root cause: Signed integer overflow (undefined behavior)

Debug and release alike: Wrong prices. At -O0 the multiply happens to wrap to a negative number; at -O3 nothing is guaranteed at all. The optimizer didn't create this bug, and testing at -O0 wouldn't have caught it either: only an input above ~21.5M does.

Why it survives: Nothing traps on signed overflow, so the bad value flows downstream silently. -fsanitize=signed-integer-overflow (part of UBSan) reports it on the first bad input.

Fix:

```
int64_t parse_price(const char* data) {
    int32_t price_scaled = read_int32(data);
    int64_t multiplier = 100; // Promote to int64_t first

    int64_t price = price_scaled * multiplier; // No overflow
    return price;
}
```

0.3 Common Undefined Behavior in Performance Code

1. Strict Aliasing Violations

The Rule: Accessing an object through a pointer of incompatible type is undefined behavior.

Common violation in low-latency code:

```
// Type punning: Reading bit representation of float
float price = 123.45f;
uint32_t bits = *(uint32_t*)&price; // UB: strict aliasing violation!
```

Why developers do this: - Fast conversion between types - Access to bit representation - Interfacing with hardware or network protocols

Why it's UB: The compiler assumes that pointers of different types don't alias (point to the same memory). This assumption enables optimizations:

```
void update(float* f, uint32_t* u) {
    *f = 1.0f;
    *u = 0x3F800000; // Compiler assumes this doesn't affect *f
    use(*f);         // Compiler may assume *f is still 1.0f!
}
```

The correct way:

Option 1: memcpy (compiler optimizes to same assembly)

```
float price = 123.45f;
uint32_t bits;
std::memcpy(&bits, &price, sizeof(bits)); // Legal, optimizes to direct access
```

Option 2: std::bit_cast (C++20)

```
float price = 123.45f;
uint32_t bits = std::bit_cast<uint32_t>(price); // Clean, explicit
```

Option 3: Union (limited cases)

```
union FloatBits {
    float f;
    uint32_t u;
};
```

```
FloatBits fb;
fb.f = 123.45f;
uint32_t bits = fb.u; // Legal in C; UB in C++ (reading an inactive member)
```

GCC and Clang document union punning as a supported extension, so it works in practice, but it isn't portable C++. Prefer `std::bit_cast` or `memcpy`.

Exception: `char`, `unsigned char` and `std::byte` glvalues can read any object (used for serialization).

The converse is not allowed: Treating a received byte buffer as a struct, `reinterpret_cast<const Header*>(buf)->seq`, is UB because no `Header` object lives there (and it may be misaligned). Use `memcpy` into a local `Header`, or C++23 `std::start_lifetime_as<Header>(buf)`. Compilers turn a fixed-size `memcpy` into a plain load, so there is no speed penalty.

2. Data Races

The Rule: If two threads access the same memory location, at least one is a write, and they're not synchronized, it's undefined behavior.

Common violation:

```
class PositionTracker {
    int position_; // Not atomic!

public:
    void add_trade(int quantity) {
        position_ += quantity; // Multiple threads call this - data race!
    }

    int get_position() const {
        return position_; // Reading while other thread writes - data race!
    }
};
```

Why it's UB: - Compiler assumes no data races, caches values in registers - CPU can reorder operations - Non-atomic read-modify-write can be interleaved

Real example:

```
// Thread 1:
position_ += 100; // Read: 0, compute: 100, write: 100

// Thread 2 (interleaved):
position_ += 200; // Read: 0, compute: 200, write: 200
```

```
// Final result: 200 (lost update!)
// Or worse: compiler optimizes based on "no data races" assumption
```

The correct way:

```
class PositionTracker {
    std::atomic<int> position_{0};

public:
    void add_trade(int quantity) {
        position_.fetch_add(quantity, std::memory_order_relaxed);
    }

    int get_position() const {
        return position_.load(std::memory_order_relaxed);
    }
};
```

Critical distinction: - volatile prevents compiler optimization (for memory-mapped I/O) - std::atomic provides synchronization between threads - **They are not interchangeable!**

3. Integer Overflow (Signed)

The Rule: Signed integer overflow is undefined behavior. Unsigned integer overflow is well-defined (wraps around).

Common violation:

```
int32_t compute_notional(int32_t shares, int32_t price_cents) {
    return shares * price_cents; // UB if result > INT32_MAX
}

// Example: 1,000,000 shares × 150,000 cents = 150,000,000,000
// INT32_MAX = 2,147,483,647
// Overflow!
```

Why it's dangerous: Compiler can assume overflow never happens:

```
bool will_overflow(int a, int b) {
    return (a + b) < a; // Trying to detect overflow
}

// Compiler reasoning:
// "Overflow is UB, so a + b is the mathematically exact sum"
// "Then (a + b) < a holds exactly when b < 0"

// Optimized to:
bool will_overflow(int a, int b) {
    return b < 0; // No longer detects overflow at all!
}
```

The correct way:

Option 1: Use wider type

```
int64_t compute_notional(int32_t shares, int32_t price_cents) {
    return static_cast<int64_t>(shares) * price_cents; // No overflow
}
```

Option 2: Use unsigned (if appropriate)

```
uint32_t compute_hash(uint32_t value) {  
    return value * 31 + 17; // Wrapping is well-defined for unsigned  
}
```

Option 3: Use compiler builtins

```
bool safe_multiply(int a, int b, int* result) {  
    return !__builtin_mul_overflow(a, b, result);  
}
```

4. Out-of-Bounds Array Access

The Rule: Accessing an array outside its bounds is undefined behavior.

```
int arr[10];  
arr[10] = 42; // UB! Valid indices are 0-9
```

Why it's dangerous:

```
int process_orders(Order* orders, size_t count) {  
    int total = 0;  
    for (size_t i = 0; i <= count; ++i) { // BUG: should be i < count  
        total += orders[i].quantity;  
    }  
    return total;  
}
```

```
// Compiler might:  
// - Assume the loop never accesses out-of-bounds  
// - Optimize based on that assumption  
// - Transform the code in ways that break your program
```

The correct way:

```
// Use correct bounds  
for (size_t i = 0; i < count; ++i) {  
    total += orders[i].quantity;  
}  
  
// Or use iterators  
for (const auto& order : orders_vector) {  
    total += order.quantity;  
}  
  
// Or use std::span (C++20)  
void process_orders(std::span<const Order> orders) {  
    for (const auto& order : orders) { /* size travels with the pointer */ }  
    // Note: span::operator[] is NOT bounds-checked by the standard.  
    // Enable library hardening (-D_GLIBCXX_ASSERTIONS for libstdc++)  
    // in test builds to get checks.  
}
```

5. Use-After-Free and Dangling References

The Rule: Accessing memory after it's been freed is undefined behavior.

```
int* ptr = new int(42);
delete ptr;
*ptr = 43; // UB! Use-after-free
```

Subtle version:

```
int& get_reference() {
    int local = 42;
    return local; // UB! Returning reference to local variable
}
```

```
void use_it() {
    int& ref = get_reference();
    std::cout << ref; // Accessing destroyed object
}
```

Dangling pointers in containers:

```
std::vector<int> vec = {1, 2, 3};
int* ptr = &vec[0];

vec.push_back(4); // May reallocate!
*ptr = 10;        // UB if reallocation happened
```

The correct way:

```
// Use RAII (std::unique_ptr, std::shared_ptr)
std::unique_ptr<int> ptr = std::make_unique<int>(42);
// Automatic cleanup, no use-after-free

// Don't return references to locals
int get_value() { // Return by value
    int local = 42;
    return local;
}

// Be careful with container iterators
std::vector<int> vec = {1, 2, 3};
// Don't hold pointers/references across modifications
```

6. Uninitialized Variables

The Rule: Reading an uninitialized variable is undefined behavior. (C++26 reclassifies reading an uninitialized *local* as “erroneous behavior”, which is still a bug but no longer lets the compiler assume anything.)

```
int x; // Uninitialized
int y = x + 5; // UB! Reading uninitialized value
```

Why debug builds hide this:

```
// -O0 (debug): Stack might be zeroed
int x; // Often 0 (by chance)

// -O3 (release): No initialization
int x; // Could be anything
```

Impact on optimizations:

```
bool is_positive() {
    int x; // Uninitialized
```

```

    if (x > 0) {
        return true;
    }
    return false;
}

// Compiler can assume x has *some* value
// But since reading it is UB, can optimize to anything!

```

The correct way:

```

int x = 0; // Always initialize
int y = x + 5; // Safe

// Or in modern C++:
int x{}; // Zero-initialization
int y{5}; // Direct initialization

```

0.4 Debug vs Release: The Behavior Gap

One of the most insidious aspects of undefined behavior is that it often “works” in debug builds but fails in release builds.

Why Debug Builds Hide Bugs

Memory initialization:

```

int x; // Uninitialized

// -O0 (debug): Stack often zeroed for debugging
// x might be 0

// -O3 (release): No initialization overhead
// x contains garbage

```

Optimizations disabled:

```

// -O0: Code executes in the order you wrote it
for (int i = 0; i < n; ++i) {
    array[i] = i;
}

// -O3: Compiler can reorder, vectorize, unroll
// UB that "works" with sequential execution can break

```

Timing differences:

```

// Data race example:
int shared_value = 0;

// Thread 1:
shared_value = 42;

// Thread 2:
if (shared_value != 0) {
    use(shared_value);
}

```

```

}

// -O0: Slow enough that thread 1 usually finishes first
// Race condition hidden by timing

// -O3: Fast enough that race is exposed
// Also, compiler might cache shared_value in register

```

Illustrative Example

Code:

```

class OrderCache {
    std::unordered_map<uint64_t, Order> orders_;

public:
    void add_order(const Order& order) {
        orders_[order.id] = order; // Not thread-safe!
    }

    Order* find_order(uint64_t id) {
        auto it = orders_.find(id);
        return it != orders_.end() ? &it->second : nullptr;
    }
};

```

Usage:

```

// Thread 1: Adding orders
cache.add_order(order);

// Thread 2: Finding orders
Order* order = cache.find_order(id);

```

Debug build (-O0): - Slow enough that races rarely manifest - Map implementation might have extra checks - Passed all tests

Release build (-O3): - Fast enough that races happen frequently - An insert that triggers a rehash frees the bucket array while the other thread is walking it - Random crashes, corrupted data, hard to reproduce under a debugger

Fix: Protect the map with a lock, or give each thread its own cache

Testing Strategy

Never rely on debug-only testing:

```

# Test with production flags
g++ -O3 -march=native -DNDEBUG test.cpp

# Use sanitizers in development
# (ThreadSanitizer cannot be combined with AddressSanitizer: two builds)
g++ -O2 -g -fsanitize=address,undefined test.cpp
g++ -O2 -g -fsanitize=thread test.cpp

```

CI/CD pipeline:

```

# Stage 1: Development (all sanitizers)
cmake -DCMAKE_BUILD_TYPE=Debug -DENABLE_SANITIZERS=ON

```

```
# Stage 2: Pre-production (release flags + sanitizers)
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DENABLE_SANITIZERS=ON

# Stage 3: Production (release flags, no sanitizers)
cmake -DCMAKE_BUILD_TYPE=Release -DENABLE_SANITIZERS=OFF
```

0.5 Optimization and the As-If Rule

Understanding how the compiler optimizes code is essential for writing high-performance code without introducing UB.

Loop Optimizations

Original code:

```
int sum = 0;
for (int i = 0; i < 1000; ++i) {
    sum += i;
}
```

What the compiler can do:

1. Closed-form solution:

```
int sum = (999 * 1000) / 2; // 499,500
```

2. Loop unrolling:

```
int sum = 0;
for (int i = 0; i < 1000; i += 4) {
    sum += i + (i+1) + (i+2) + (i+3);
}
// Fewer loop iterations, more work per iteration
```

3. Vectorization (SIMD):

```
// Compiler generates AVX2 instructions
// Processes 8 integers at once
```

Function Inlining

Original:

```
inline int square(int x) {
    return x * x;
}
```

```
int area = square(5);
```

After inlining:

```
int area = 25; // Computed at compile time!
```

Impact on HFT:

```
// Hot path function
inline double compute_fair_value(double bid, double ask) {
    return (bid + ask) / 2.0;
}
```

```
// Call site:
double fair = compute_fair_value(100.5, 100.7);

// Optimized to:
double fair = 100.6; // No function call overhead!
```

Dead Store Elimination

Original:

```
int x = 5; // First assignment
x = 10; // Second assignment
use(x);
```

Optimized:

```
int x = 10; // First assignment removed (dead store)
use(x);
```

Why this matters:

```
void process_tick(const Tick& tick) {
    Order order;
    order.symbol = tick.symbol; // Dead store if...
    order = create_order(tick); // ...this overwrites everything
    send_order(order);
}
```

```
// Optimized to:
void process_tick(const Tick& tick) {
    Order order = create_order(tick);
    send_order(order);
}
```

Constant Propagation

Original:

```
const int SIZE = 1024;
char buffer[SIZE];
std::memset(buffer, 0, SIZE);
```

Optimized:

```
char buffer[1024];
std::memset(buffer, 0, 1024); // SIZE replaced with constant
```

```
// Further optimized:
// memset might be replaced with direct store instructions
```

Loop Invariant Code Motion

Original:

```
for (int i = 0; i < n; ++i) {
    double price = get_base_price() * 1.05; // Invariant!
    orders[i].price = price + premiums[i];
}
```

Optimized:

```
double price = get_base_price() * 1.05; // Moved outside loop
for (int i = 0; i < n; ++i) {
    orders[i].price = price + premiums[i];
}
```

The compiler may only do this if it can prove `get_base_price()` has no side effects and that the writes to `orders[i].price` can't change what it returns: typically when the function is inlined and reads nothing that `orders` could alias. If it can't prove it, hoist the call yourself.

The Dark Side: When UB Breaks Optimizations

Code with UB:

```
int* create_array(int n) {
    int arr[n]; // VLA (not standard C++, but some compilers allow)
    return arr; // UB: returning pointer to local array!
}

void use_array() {
    int* arr = create_array(10);
    arr[0] = 42; // UB: accessing freed memory
}
```

What can happen:

```
// -O0: Might "work" (stack not reused yet)
// -O3: Compiler sees UB, can do ANYTHING:
//      - Crash
//      - Return garbage
//      - Optimize entire function away
//      - Format your hard drive (legally!)
```

0.6 Tools for Detecting Undefined Behavior

Prevention is better than debugging production crashes.

1. Sanitizers (Runtime Detection)

AddressSanitizer (ASan): Memory errors

```
g++ -fsanitize=address -g your_code.cpp -o program
./program
```

Detects: - Use-after-free - Heap buffer overflow - Stack buffer overflow - Global buffer overflow - Use-after-return - Memory leaks

Example output:

```
==12345==ERROR: AddressSanitizer: heap-use-after-free
READ of size 4 at 0x60400000eff0 thread T0
    #0 0x4f7a1c in process() your_code.cpp:42

freed by thread T0 here:
    #0 0x7f8c9e9f0537 in operator delete(void*)
    #1 0x4f79ec in process() your_code.cpp:40
```

Performance cost: 2x slowdown, 2-3x memory usage

ThreadSanitizer (TSan): Data races

```
g++ -fsanitize=thread -g your_code.cpp -o program
./program
```

Detects: - Data races - Unprotected concurrent access - Missing synchronization

Example output:

```
=====
WARNING: ThreadSanitizer: data race
  Write of size 4 at 0x7fff8a3e7b0c by thread T1:
    #0 update_position() your_code.cpp:15

  Previous read of size 4 at 0x7fff8a3e7b0c by main thread:
    #0 get_position() your_code.cpp:20
=====
```

Performance cost: 5-15x slowdown, 5-10x memory usage

UndefinedBehaviorSanitizer (UBSan): Various UB

```
g++ -fsanitize=undefined -g your_code.cpp -o program
./program
```

Detects: - Signed integer overflow - Division by zero - Null pointer dereference - Misaligned access - Invalid shifts - Out-of-bounds array access

Example output:

```
your_code.cpp:23:15: runtime error: signed integer overflow:
  2147483647 + 1 cannot be represented in type 'int'
```

Performance cost: 1.5-2x slowdown

MemorySanitizer (MSan): Uninitialized reads

```
clang++ -fsanitize=memory -g your_code.cpp -o program
./program
```

Detects: - Use of uninitialized memory

Note: Clang only (GCC has no MSan), and requires recompiling all dependencies, including the C++ standard library

2. Static Analysis

Clang-Tidy:

```
clang-tidy your_code.cpp -- -std=c++20 -I/path/to/includes
```

Checks: - Bugprone patterns - Performance issues - Modernization opportunities - Readability problems

Example configuration (.clang-tidy):

```
Checks: '-*,
  bugprone-*,
  performance-*,
  concurrency-*,
  cppcoreguidelines-*'
WarningsAsErrors: '*'
```

Cppcheck:

```
cppcheck --enable=all --std=c++20 your_code.cpp
```

Detects: - Null pointer dereferences - Memory leaks - Uninitialized variables - Out-of-bounds access

Clang Static Analyzer:

```
scan-build make
```

More thorough than clang-tidy, slower

3. Compiler Warnings

Enable all warnings:

```
g++ -Wall -Wextra -Wpedantic -Werror \
    -Wshadow -Wnon-virtual-dtor -Wold-style-cast \
    -Wcast-align -Wunused -Woverloaded-virtual \
    -Wconversion -Wsign-conversion -Wformat=2 \
    your_code.cpp
```

Key flags: - `-Wall -Wextra`: Most common warnings - `-Werror`: Treat warnings as errors (CI/CD) - `-Wpedantic`: Strict standard compliance - `-Wshadow`: Variable shadowing - `-Wconversion`: Implicit conversions

4. Runtime Assertions

In development:

```
#include <cassert>

void process_order(const Order* order) {
    assert(order != nullptr); // Debug builds only
    assert(order->quantity > 0);

    // ... process ...
}
```

In production (with custom handling):

```
// do { } while (0) makes the macro a single statement, so
// `if (x) VERIFY(y); else ...` binds the else correctly
#define VERIFY(condition) \
    do { \
        if (!(condition)) [[unlikely]] { \
            log_error(__FILE__, __LINE__, #condition); \
            emergency_shutdown(); \
        } \
    } while (0)

void process_order(const Order* order) {
    VERIFY(order != nullptr);
    VERIFY(order->quantity > 0);

    // ... process ...
}
```

5. Fuzzing

LibFuzzer (continuous random testing):

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    if (size < 8) return 0;

    // Test your parser/processor
    parse_market_data(data, size);

    return 0;
}

clang++ -fsanitize=fuzzer,address your_fuzzer.cpp -o fuzzer
./fuzzer
```

Finds edge cases that unit tests miss

Development Workflow

Recommended setup:

```
# Development build (all sanitizers)
cmake -DCMAKE_BUILD_TYPE=Debug \
      -DCMAKE_CXX_FLAGS="-fsanitize=address,undefined -g" \
      .

# Testing build (thread sanitizer separate - incompatible with others)
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo \
      -DCMAKE_CXX_FLAGS="-fsanitize=thread -g" \
      .

# Production build (no sanitizers, maximum optimization)
cmake -DCMAKE_BUILD_TYPE=Release \
      -DCMAKE_CXX_FLAGS="-O3 -march=native -DNDEBUG" \
      .
```

CI Pipeline:

```
stages:
  - name: static_analysis
    run: clang-tidy src/**/*.cpp

  - name: unit_tests_asan
    build_flags: -fsanitize=address,undefined
    run: ./run_tests

  - name: unit_tests_tsan
    build_flags: -fsanitize=thread
    run: ./run_tests

  - name: integration_tests
    build_flags: -O3 -march=native
    run: ./run_integration_tests

  - name: performance_tests
    build_flags: -O3 -march=native -DNDEBUG
    run: ./run_benchmarks
```

0.7 Case Studies: Common UB Bug Patterns

These are illustrative composites of well-known bug classes, not reports of specific incidents.

Case Study 1: The Disappearing Null Check

System: Order router

Code (after a refactor moved the price read to the top):

```
void route_order(Order* order) {
    const int64_t price = order->price;  // Dereference!

    // Defensive null check - now AFTER the dereference
    if (order == nullptr) {
        log_error("Null order received");
        return;
    }

    // Validate order
    if (price <= 0) {
        log_error("Invalid price");
        return;
    }

    // Route order
    send_to_exchange(order);
}
```

Root cause:

```
// Compiler reasoning:
// "order->price was already read, so order != nullptr (else UB)"
// "Therefore the null check is dead code"

// Optimized to:
void route_order(Order* order) {
    if (order->price <= 0) {
        log_error("Invalid price");
        return;
    }
    send_to_exchange(order);  // Null check and its log line removed!
}
```

What happened: An upstream bug occasionally passed a null pointer. The process crashed with no “Null order received” line in the log, so the investigation started in the wrong place. The check looked present in the source, but the compiler had legally deleted it.

Fix:

```
void route_order(Order* order) {
    // Check before any dereference
    if (order == nullptr || order->price <= 0) {
        log_error("Invalid order");
        return;
    }
    send_to_exchange(order);
}
```

```

// Or better: Use references (cannot be null)
void route_order(const Order& order) {
    if (order.price <= 0) {
        log_error("Invalid price");
        return;
    }
    send_to_exchange(order);
}

```

Case Study 2: The Overflow That Wasn't Checked

System: Position risk calculator

Impact: A trade that should be rejected is accepted

Code:

```

bool check_position_limit(int32_t current_position, int32_t trade_size) {
    int32_t new_position = current_position + trade_size;

    if (new_position > MAX_POSITION_LIMIT) {
        return false; // Reject trade
    }
    return true; // Accept trade
}

```

What happened:

```

// Scenario:
current_position = 2,000,000,000 (2 billion)
trade_size      = 200,000,000 (200 million)
MAX_POSITION_LIMIT = 2,000,000,000

// Expected: new_position = 2,200,000,000, reject trade
// Actual: Overflow! (UB)

// In practice the add wraps: new_position = -2,094,967,296
// -2,094,967,296 > MAX_POSITION_LIMIT is false, so the trade is accepted

```

Root cause: Signed integer overflow is UB. This one fails in every build: at -O0 the wrapped value slips past the check, and at -O3 the compiler is not even obliged to produce that wrapped value. The check has to be done before the addition, or in a wider type.

Fix:

```

bool check_position_limit(int32_t current_position, int32_t trade_size) {
    // Check for overflow before it happens
    if (trade_size > 0 && current_position > MAX_POSITION_LIMIT - trade_size) {
        return false; // Would overflow
    }
    if (trade_size < 0 && current_position < MIN_POSITION_LIMIT - trade_size) {
        return false; // Would underflow
    }

    int32_t new_position = current_position + trade_size;
    if (new_position > MAX_POSITION_LIMIT) {
        return false;
    }
}

```

```

    return true;
}

// Or use __builtin_add_overflow:
bool check_position_limit(int32_t current_position, int32_t trade_size) {
    int32_t new_position;
    if (__builtin_add_overflow(current_position, trade_size, &new_position)) {
        return false; // Overflow detected
    }
    return new_position >= MIN_POSITION_LIMIT &&
           new_position <= MAX_POSITION_LIMIT;
}

// Or simplest: compute in a wider type
bool check_position_limit(int32_t current_position, int32_t trade_size) {
    const int64_t new_position = int64_t{current_position} + trade_size;
    return new_position >= MIN_POSITION_LIMIT &&
           new_position <= MAX_POSITION_LIMIT;
}

```

Case Study 3: The Data Race in Position Tracking

System: Real-time position tracker

Impact: Intermittent incorrect positions, occasional crashes

Code:

```

class PositionManager {
    std::unordered_map<std::string, int> positions_;

public:
    void update_position(const std::string& symbol, int delta) {
        positions_[symbol] += delta; // Multiple threads!
    }

    int get_position(const std::string& symbol) {
        return positions_[symbol]; // operator[] INSERTS if missing - a write!
    }
};

```

What happened: - Multiple threads called `update_position` concurrently - Even `get_position` mutates the map: `operator[]` inserts missing symbols - `+=` on the same entry from two threads loses updates - An insert that triggers a rehash reallocates the bucket array while another thread is traversing it → corrupted map, crash

Debug build: Rarely manifested (timing)

Release build: Frequent corruption

Root cause: Data race (UB). The unsynchronized container is the problem, not register caching: `std::unordered_map` gives no guarantees for concurrent writes, or for a read concurrent with a write.

Fix:

```

class PositionManager {
    std::unordered_map<std::string, int64_t> positions_;
    mutable std::mutex mutex_;

public:

```

```

void update_position(const std::string& symbol, int64_t delta) {
    std::lock_guard lock(mutex_);
    positions_[symbol] += delta;
}

int64_t get_position(const std::string& symbol) const {
    std::lock_guard lock(mutex_);
    auto it = positions_.find(symbol); // find, not operator[]
    return it != positions_.end() ? it->second : 0;
}
};

// Lower latency: map symbols to dense integer IDs once at startup and keep
// std::array<std::atomic<int64_t>, MAX_SYMBOLS> positions_;
// The hot path is then a relaxed fetch_add with no lock, no hashing of
// strings and no insertion (which is what made the map unsafe).

```

Case Study 4: The Strict Aliasing Bug

System: Binary market data parser (e.g. an ITCH-style feed with big-endian fields at arbitrary offsets)

Impact: Random message corruption

Code:

```

// Fast conversion from network byte order
uint32_t parse_uint32(const char* buffer) {
    // Cast char* to uint32_t* and dereference
    uint32_t value = *(uint32_t*)buffer; // UB: strict aliasing + misalignment!
    return ntohl(value);
}

void parse_message(const char* msg) {
    // ... parse fields ...
    uint32_t sequence = parse_uint32(msg + offset);
    // ... use sequence ...
}

```

What happened: - Worked for months on x86, which tolerates unaligned loads - Then the receive path was changed to copy packets into the parse buffer with 8-byte `uint64_t` stores - After inlining, the compiler moved the `uint32_t` load above those stores and read stale bytes

Root cause: Two kinds of UB in one line - **Strict aliasing:** the compiler assumes a `uint32_t` load cannot observe a `uint64_t` store, so it may reorder them - **Alignment:** `msg + offset` is rarely 4-byte aligned. x86 forgives it, but the compiler may emit instructions that require alignment, and other architectures trap (SIGBUS)

Fix:

```

uint32_t parse_uint32(const char* buffer) {
    uint32_t value;
    std::memcpy(&value, buffer, sizeof(value)); // Legal!
    return ntohl(value);
}

// Compiler optimizes memcpy to a single unaligned load (same assembly on x86)
// But it's legal and correct: no aliasing or alignment assumptions
// C++23: return std::byteswap(value) on little-endian targets instead of ntohl

```

0.8 The Optimization Paradox

The Tension

In high-frequency trading, we face a fundamental paradox:

We need aggressive optimization: - `-O3`: Maximum optimization - `-march=native`: Use all CPU features
- `-flto`: Link-time optimization across entire program - Every nanosecond counts

But aggressive optimization exposes UB: - Removes null checks that follow a dereference - Assumes no integer overflow - Assumes no data races - Reorders memory accesses

And we can't test with `-O0`: - Different behavior (timing, layout) - Different code paths (inlining, unrolling) - Different bugs manifest

The Solution: Correct First, Fast Second

Step 1: Write Correct Code - Avoid all undefined behavior - Use sanitizers during development - Follow strict coding standards - Peer review for UB patterns

Step 2: Verify Correctness

```
# Run all tests with sanitizers
make test CXXFLAGS="-fsanitize=address,undefined"
make test CXXFLAGS="-fsanitize=thread"
```

Step 3: Optimize

```
# Build with production flags
make CXXFLAGS="-O3 -march=native -DNDEBUG"
```

Step 4: Re-Verify

```
# Run tests again with production flags
make test CXXFLAGS="-O3 -march=native"
```

Step 5: Profile and Iterate

```
# Profile to find hotspots
perf record ./program
perf report
```

```
# Optimize hotspot
# Re-verify with sanitizers
# Measure improvement
```

Remember

Fast broken code is still broken.

Slow correct code can be made fast.

Fast broken code is exponentially harder to debug.

In HFT specifically: - Broken code trading at full speed can lose large amounts in minutes (Knight Capital lost \$440M in 45 minutes in 2012 to a deployment error, not UB, but the speed of loss is the point)
- Debugging production UB is nearly impossible (can't reproduce) - Prevention is 1000x cheaper than fixing

0.9 Compiler Flags for Development and Production

Development Flags

Debug build:

```
g++ -std=c++20 \  
    -O0 -g \  
    -Wall -Wextra -Wpedantic -Werror \  
    -fsanitize=address,undefined \  
    -D_GLIBCXX_DEBUG \  
    your_code.cpp -o program_debug
```

Flags explained: - -O0: No optimization (debuggable) - -g: Debug symbols - -Wall -Wextra -Wpedantic: All warnings - -Werror: Warnings are errors - -fsanitize=address,undefined: Runtime UB detection - -D_GLIBCXX_DEBUG: STL debug mode (bounds and iterator checking). It changes the ABI of standard containers, so every library that passes containers across its interface must be built with it too. -D_GLIBCXX_ASSERTIONS gives cheap bounds checks without the ABI change.

Thread testing:

```
g++ -std=c++20 \  
    -O2 -g \  
    -fsanitize=thread \  
    your_code.cpp -o program_tsan
```

Production Flags

Release build:

```
g++ -std=c++20 \  
    -O3 \  
    -march=native \  
    -flto \  
    -DNDEBUG \  
    -fno-exceptions \  
    -fno-rtti \  
    your_code.cpp -o program_release
```

Flags explained: - -O3: Maximum optimization - -march=native: Use all instructions of the *build* machine's CPU (AVX2, etc.; implies -mtune=native). Only safe if you build on the same CPU model you deploy to. Otherwise name the target explicitly (e.g. -march=icelake-server), or the binary can die with SIGILL on older production hosts. - -flto: Link-time optimization (cross-file inlining) - -DNDEBUG: Disable asserts - -fno-exceptions: No exception handling overhead (if not used) - -fno-rtti: No runtime type information (if not used)

Performance profiling build:

```
g++ -std=c++20 \  
    -O3 \  
    -march=native \  
    -g \  
    -fno-omit-frame-pointer \  
    your_code.cpp -o program_profile
```

Flags explained: - -g: Debug symbols for profiler - -fno-omit-frame-pointer: Better stack traces in profiler

CMake Configuration

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.15)
project(HFTSystem CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Common flags (a string, not a list: a list would put semicolons
# into the CMAKE_CXX_FLAGS_* strings below)
set(WARNING_FLAGS "-Wall -Wextra -Wpedantic -Werror")

# Debug build
set(CMAKE_CXX_FLAGS_DEBUG "${WARNING_FLAGS} -O0 -g -fsanitize=address,undefined")

# Release build
set(CMAKE_CXX_FLAGS_RELEASE "${WARNING_FLAGS} -O3 -march=native -DNDEBUG")

# RelWithDebInfo (for profiling)
set(CMAKE_CXX_FLAGS_RELWITHDEBINFO "${WARNING_FLAGS} -O3 -march=native -g -fno-omit-frame-pointer")

# Thread sanitizer build (custom build type: cmake -DCMAKE_BUILD_TYPE=Tsan)
set(CMAKE_CXX_FLAGS_TSAN "${WARNING_FLAGS} -O2 -g -fsanitize=thread")
set(CMAKE_EXE_LINKER_FLAGS_TSAN "-fsanitize=thread")
```

Summary

Key Takeaways

1. **C++ is defined by an abstract machine**, not specific assembly
 - Compiler can transform code preserving observable behavior
 - Enables aggressive optimization
2. **Undefined behavior means “no guarantees”**
 - Compiler assumes UB never happens
 - Can optimize based on that assumption
 - Results in “works in debug, crashes in release”
3. **Common UB in performance code:**
 - Strict aliasing violations (type punning)
 - Data races (non-atomic concurrent access)
 - Signed integer overflow
 - Out-of-bounds array access
 - Use-after-free / dangling references
4. **Debug builds hide UB** through:
 - Memory initialization
 - Disabled optimizations
 - Different timing
 - → Must test with production flags!
5. **Use tools to detect UB:**
 - AddressSanitizer (memory errors)
 - ThreadSanitizer (data races)
 - UndefinedBehaviorSanitizer (various UB)
 - Static analysis (clang-tidy, cppcheck)

- Compiler warnings (`-Wall -Wextra -Werror`)
6. **The optimization paradox:**
- Need `-O3` for performance (HFT requirement)
 - `-O3` exposes UB mercilessly
 - Solution: Correct first, fast second

Checklist Before Optimizing

Before applying any optimization technique from Chapters 1-5:

- Code is free of undefined behavior
- All tests pass with `-fsanitize=address,undefined`
- All tests pass with `-fsanitize=thread`
- Static analysis shows no issues
- Tests pass with `-O3 -march=native`
- Code review completed

Only then: - Profile to find hotspot (Chapter 4) - Apply appropriate optimization technique - Re-verify all above checks - Measure actual improvement

What's Next

Chapter 1 covers memory management and cache behaviour, the first of the HFT-specific topics. Now that you understand how to write correct C++ code that can be aggressively optimized, you're ready to learn the HFT-specific techniques that achieve nanosecond-level performance.

The foundation is critical: Everything in the following chapters assumes your code is free of undefined behavior. Fast broken code is worthless in HFT—it loses money.

Further Reading

Essential

C++ Standards and References: - C++ Standard (ISO/IEC 14882), Section 6.9: Program execution - C++ Reference: Undefined Behavior

Books: - “C++ Concurrency in Action” (Anthony Williams) — Chapter 5: Memory Model - “Effective Modern C++” (Scott Meyers) — Items on UB and optimization

Articles: - “Understanding Strict Aliasing” (Mike Acton) - “What Every Programmer Should Know About Undefined Behavior” (Chris Lattner)

Deep Dive

Compiler Internals: - “A Guide to Undefined Behavior in C and C++” (John Regehr) - LLVM Blog: Undefined Behavior posts - GCC Wiki: Undefined Behavior

Tools: - Compiler Explorer — See what compiler generates - C++ Insights — See compiler transformations - Quick Bench — Micro-benchmarking

Talks: - CppCon: “Undefined Behavior is Not an Error” (Barbara Geller, Ansel Sermersheim) - CppCon: “The Speed of Concurrency” (Fedor Pikus) - CppCon: “Type Punning in C++17” (Scott Schurr)

Ready to proceed? → Chapter 1: Memory Management & Performance

Back to the overview → README

Low-Latency C++ Senior Developer Interview Preparation

Interview Details

Position: Low-Latency C++ Senior Developer

Focus Areas: High-frequency trading systems, low-latency architecture, modern C++

Chapters

This preparation package contains six chapters covering the critical topics for a low-latency C++ position:

0. The C++ Abstract Machine

- The as-if rule and observable behavior
 - Undefined behavior common in performance code (strict aliasing, data races, signed overflow, lifetime bugs)
 - Why bugs appear only in release builds
 - Sanitizers, static analysis and compiler warnings
-

1. Memory Management & Performance

- Cache-friendly data structures and false sharing
 - Memory pools and custom allocators (Pool, Arena, Thread-local)
 - Compiler optimizations (branch prediction, inlining, prefetching)
-

2. Concurrency & Threading

- Lock-free data structures: SPSC, MPSC, SPMC (work distribution and broadcast) and MPMC queues, stack, hash map
 - Memory ordering deep dive (relaxed, acquire/release, seq_cst)
 - Thread pinning and CPU affinity for deterministic performance
 - Context switch avoidance (busy-wait vs sleep)
 - Synchronization primitives (spinlocks, seqlocks, RWLocks)
-

3. Modern C++ Features

- Move semantics and perfect forwarding for zero-copy operations
 - Template metaprogramming for compile-time optimization
 - constexpr for zero-runtime-cost computations
 - std::variant and std::optional for type-safe, efficient code
 - Smart pointers - when to use and when to avoid
 - Ranges and views (C++20) for allocation-free pipelines
 - Concepts (C++20) for better compile-time checking
-

4. Low-Latency Techniques & Best Practices

- Busy-waiting vs sleeping strategies
- Kernel bypass (DPDK, Solarflare, memory-mapped I/O)
- System call minimization and batching
- High-resolution timing (TSC, HPET, hardware timestamps)
- TCP/UDP optimization techniques
- SIMD and vectorization for parallel processing
- Profiling and measurement tools

5. Trading System Components & Measurement

- Order book design (market-by-order to price levels, data structure choices)
 - Binary exchange protocols (ITCH, MoldUDP64, SBE, fast FIX parsing)
 - Cache warming for rarely-executed order paths
 - Microbenchmarking pitfalls (dead code elimination, throughput vs latency, hot caches)
-

Quick Reference: Performance Numbers

Latency Hierarchy (Approximate)

L1 cache hit:	4-5 cycles	(~1-2 ns)
L2 cache hit:	12-20 cycles	(~3-7 ns)
L3 cache hit:	40-75 cycles	(~10-20 ns)
RAM access:	200-300+ cycles	(~60-100 ns)
Branch mispredict:	15-20 cycles	(~5 ns)
rdtsc:	~25 cycles	(~6-8 ns)
clock_gettime (vDSO):	~20 ns	(no system call)
System call (trivial):	~100 ns; 200-500 ns	with Spectre/Meltdown mitigations
Context switch:	1-10 s	(direct cost + cache/TLB refill)
Minor page fault:	~1 s	(first touch of a page)
Major page fault:	~100 s (NVMe) to ~10 ms	(HDD)
NVMe read:	~10-100 s	
Network RTT (colo LAN, kernel):	~10-40 s	
Network RTT (colo LAN, bypass):	~2-5 s	
Network RTT (Internet):	10-100+ ms	

Memory & Cache

Cache line size:	64 bytes (Intel prefetches in 128-byte pairs)
False sharing penalty:	10-100x slowdown on the contended variable
Cross-core cache line transfer:	~50-100 ns
malloc/free (fast path):	~20 ns; microseconds when the allocator hits the OS
Memory pool allocation:	~5 ns

Threading

Thread creation:	~10-50 s
Atomic RMW (uncontended):	~5-10 ns (~20 cycles)
Atomic RMW (contended):	~50-100+ ns (cache line bounces between cores)
Spin lock/mutex (uncontended):	~20 ns (both a single atomic operation)
Mutex (contended, sleeps):	1-10 s (futex wake-up + context switch)
PAUSE instruction:	~10 cycles (pre-Skylake) to ~140 cycles (Skylake+)

Network

Kernel network stack:	~5-20 s per packet (one way)
Kernel bypass:	~0.5-2 s per packet (one way)
TCP retransmission timeout:	200 ms on Linux (why packet loss hurts)

These are order-of-magnitude figures for current x86 servers; exact values vary by CPU generation and configuration. Quote them as ranges, and say what you'd measure.

Common Interview Topics Matrix

Topic	Importance	Difficulty	Common Questions
Cache optimization		Medium	False sharing, data layout
Lock-free programming		Hard	ABA problem, memory ordering
Memory management		Medium	Custom allocators, pools
Thread pinning		Medium	CPU affinity, NUMA
Move semantics		Easy	When and how to use
Kernel bypass		Hard	DPDK, use cases
SIMD		Medium	AVX2, when to use
Profiling		Medium	perf, VTune
Network protocols		Easy	TCP vs UDP, FIX
Order book design		Medium	Data structures, best bid/ask updates
Binary protocols		Medium	ITCH/SBE parsing, fixed-point prices
Measurement		Medium	Microbenchmark pitfalls, cold vs hot paths
constexpr		Easy	Compile-time computation

Pre-Interview Checklist

Technical Review

- ☐ Review the interview Q&A at the end of each chapter
- ☐ Understand cache line size and alignment
- ☐ Know memory ordering semantics (relaxed, acquire/release, seq_cst)
- ☐ Can implement SPSC queue from memory
- ☐ Understand ABA problem and solutions
- ☐ Know when to use move semantics
- ☐ Familiar with TSC-based timing
- ☐ Can explain kernel bypass benefits/tradeoffs
- ☐ Can design an order book and justify the data structures
- ☐ Can explain why a microbenchmark and production disagree

Practical Preparation

- ☐ Prepare 2-3 examples from your experience optimizing code
- ☐ Have specific latency numbers ready (before/after optimization)
- ☐ Prepare questions about Barclays' tech stack
- ☐ Review C++11/14/17/20 features
- ☐ Practice explaining technical concepts clearly

Questions to Ask Interviewer

Technical: - What latency targets does the team work with? (microseconds? nanoseconds?) - What hardware setup? (FPGA involvement? Dedicated cores?) - Network infrastructure? (Kernel bypass? Which technology?) - Languages beyond C++? (Python for tools, etc.) - How do you test latency-critical changes?

Team & Process: - Team structure and collaboration with traders/quants - Typical development workflow and code review process - Deployment process for production systems - On-call expectations and incident response - Learning and development opportunities

Domain: - Which asset classes? (Equities, FX, Fixed Income?) - Type of trading? (Market making, execution, arbitrage?) - Market data feed sources and protocols? - Regulatory requirements and constraints?

Key Principles to Emphasize

1. Measurement First

“You can’t optimize what you can’t measure.” - Always profile before optimizing - Focus on tail latency (p99, p99.9), not just average - Use proper instrumentation (TSC, perf, VTune)

2. Understand Trade-offs

- Latency vs Throughput
- CPU usage vs Latency (busy-wait costs CPU)
- Code complexity vs Performance gains
- Portability vs Hardware-specific optimizations

3. Hot Path vs Cold Path

- Optimize hot paths aggressively
- Keep cold paths simple and maintainable
- Don’t premature optimize

4. Predictability Matters

- Consistent latency > Fast average with spikes
- Avoid allocations, locks, system calls in hot path
- Pin threads, isolate cores, lock memory

5. Modern C++ Philosophy

- Zero-cost abstractions
 - Pay only for what you use
 - Compile-time over runtime
-

Interview Day Strategy

Before the Interview

1. Review the interview Q&A at the end of each chapter
2. Prepare your optimization war stories with specific numbers
3. Get good sleep - technical interviews are mentally demanding
4. Arrive 15 minutes early to settle nerves

During the Interview

1. **Listen carefully** to the question before answering
2. **Think out loud** - explain your reasoning
3. **Ask clarifying questions** - about constraints, scale, requirements
4. **Start simple, then optimize** - show you understand trade-offs
5. **Use concrete examples** - especially from your experience
6. **Be honest** about what you don’t know - offer to reason through it

Behavioral Questions

Prepare STAR (Situation, Task, Action, Result) stories for: - Debugging a critical production issue - Optimizing a slow system (with latency numbers!) - Working under pressure (market hours) - Disagreeing with a design decision - Learning a new technology quickly

Additional Resources

Tools to Mention

- **Profiling:** perf, VTune, flamegraphs
 - **Debugging:** gdb, ThreadSanitizer, Helgrind
 - **Network:** tcpdump, Wireshark
 - **Benchmarking:** Google Benchmark, custom instrumentation
-

This preparation guide was created specifically for a Senior Low-Latency C++ Developer position. All six chapters contain comprehensive explanations, code examples, and interview questions with answers.

Building PDFs

```
nix build          # PDFs land in ./result/
nix develop        # shell with pandoc + TeX Live
```

Contents

Chapter 1: Memory Management & Performance	1
Table of Contents	1
Cache-Friendly Data Structures	1
Cache Line Basics	1
False Sharing - Critical Issue	1
Data Structure Layout	3
Array of Structures vs Structure of Arrays	3
Memory Pools & Custom Allocators	4
Why Custom Allocators?	4
Basic Pool Allocator	4
Thread-Local Pools	5
Arena Allocator (Bump Allocator)	5
Compiler Optimizations	7
Branch Prediction Hints	7
Inlining	8
Prefetching	8
Interview Questions & Answers	8
1. "Explain the difference between L1, L2, and L3 cache and their latencies."	8
2. "What is false sharing and how do you avoid it?"	9
3. "Why would you avoid using std::vector in a low-latency system?"	9
Practical Exercise	11

Chapter 1: Memory Management & Performance

Table of Contents

1. Cache-Friendly Data Structures
 2. Memory Pools & Custom Allocators
 3. Compiler Optimizations
 4. Interview Questions & Answers
-

Cache-Friendly Data Structures

Cache Line Basics

- Modern CPUs have cache lines of 64 bytes (typically)
- When you access one byte, the entire cache line loads
- L1 cache: ~1-4 cycles, L2: ~10-20 cycles, L3: ~40-75 cycles, RAM: ~200+ cycles

False Sharing - Critical Issue

BAD: False sharing

```
struct Counters {  
    std::atomic<int64_t> thread1_count; // Bytes 0-7  
    std::atomic<int64_t> thread2_count; // Bytes 8-15  
    // Both in same 64-byte cache line!  
};
```

```
Counters counters;
```

```
// Thread 1
```

```

void thread1_work() {
    for (int i = 0; i < 1000000; ++i) {
        counters.thread1_count.fetch_add(1, std::memory_order_relaxed);
    }
}

// Thread 2
void thread2_work() {
    for (int i = 0; i < 1000000; ++i) {
        counters.thread2_count.fetch_add(1, std::memory_order_relaxed);
        // Cache line ping-pongs between cores!
    }
}

```

Why this is bad: Even though `thread1_count` and `thread2_count` are separate variables accessed by different threads, they reside on the same 64-byte cache line. When Thread 1 writes to `thread1_count`, the cache coherency protocol (MESI/MOESI) invalidates that entire cache line in Thread 2's cache. Thread 2 must then reload the cache line to access `thread2_count`, even though `thread2_count` hasn't changed. This creates false "sharing" - the threads aren't actually sharing data, but the hardware treats them as if they are. The result is constant cache line bouncing between CPU cores, turning what should be independent operations into serialized, expensive memory operations.

Detection: - Use `perf` on Linux: `perf c2c record ./program` to detect cache-to-cache transfers - Use Intel VTune's "Memory Access Analysis" - Look for high L3 cache miss rates with multiple threads

GOOD: Cache line alignment

```

struct Counters {
    alignas(64) std::atomic<int64_t> thread1_count;
    alignas(64) std::atomic<int64_t> thread2_count;
    // alignas also rounds sizeof(Counters) up to 128, so no manual padding needed
};

// C++17 names the constant (check your compiler supports it):
#include <new>
struct Counters17 {
    alignas(std::hardware_destructive_interference_size) std::atomic<int64_t> thread1_count;
    alignas(std::hardware_destructive_interference_size) std::atomic<int64_t> thread2_count;
};

```

Intel's L2 "adjacent line" prefetcher pulls cache lines in 128-byte pairs, so some teams align hot per-thread data to 128 bytes rather than 64.

Why this is good: With `alignas(64)`, `thread1_count` and `thread2_count` live on separate cache lines. Now when Thread 1 writes to `thread1_count`, it only invalidates its own cache line - Thread 2's cache line containing `thread2_count` remains valid and hot in its CPU core's cache. The threads can now operate truly independently without cache coherency traffic between cores. We trade a small amount of memory (padding bytes) for dramatic performance improvements when multiple threads access the structure concurrently.

ALTERNATIVE: Thread-local storage

```

// Best: Avoid sharing altogether
thread_local int64_t my_counter = 0;

// Aggregate later if needed
std::atomic<int64_t> total{0};
// Each thread updates only at end
total.fetch_add(my_counter);

```

Data Structure Layout

BAD: Poor cache locality

```
struct Trade {
    std::string symbol;      // Pointer to heap
    double price;
    std::string counterparty; // Another pointer
    int64_t timestamp;
};
```

Why this is bad: Each `std::string` is actually a small object containing a pointer to heap-allocated character data. When you access a `Trade` object, the CPU loads the struct into cache, but the actual string data lives elsewhere in memory. Accessing `trade.symbol` requires a second memory fetch (pointer chase) which likely causes a cache miss. Even worse, these heap allocations are scattered throughout memory, destroying spatial locality. Processing an array of trades means constant cache misses as you jump between the struct locations and their string data locations.

BETTER: Keep hot data together

```
struct Trade {
    double price;           // Hot data first
    int64_t timestamp;
    uint32_t symbol_id;     // Use ID instead of string
    uint32_t counterparty_id;
    // Cold data below or in separate structure
};
```

Why this is better: By using integer IDs instead of strings, all the data is stored inline within the struct - no heap allocations, no pointer chasing. The entire `Trade` object fits in 24 bytes and can be loaded into cache in a single memory operation. When processing an array of trades, the CPU's hardware prefetcher can efficiently stream data from memory because trades are contiguous and predictable. Symbol lookup (if needed) happens separately via a hash table or array indexed by `symbol_id`, keeping the hot path (processing trades) fast and cache-friendly. We've traded a bit of convenience (direct string access) for 10-100x better performance in tight loops.

Array of Structures vs Structure of Arrays

AoS - Bad for some access patterns

```
struct Tick {
    double price;
    int64_t timestamp;
    uint32_t volume;
};
std::vector<Tick> ticks;
```

Why this can be bad: When you only need to process prices (e.g., calculating average price), you still load entire `Tick` structs into cache - wasting 2/3 of the cache bandwidth on timestamp and volume data you don't need. The CPU's SIMD units can't efficiently vectorize operations because the prices aren't contiguous in memory - they're interleaved with other fields. For operations that access only one or two fields, you're constantly evicting useful data from cache to make room for data you don't need.

SoA - Better for SIMD, better cache usage when processing prices

```
struct TickData {
    std::vector<double> prices;
    std::vector<int64_t> timestamps;
};
```

```
std::vector<uint32_t> volumes;
};
```

Why this is better: All prices are packed contiguously in memory, so when you need to process prices, you load only price data - maximizing cache bandwidth efficiency. SIMD operations can process 4-8 prices at once because they're sequential in memory (no interleaved data to skip). The hardware prefetcher loves this pattern and can aggressively fetch ahead. When you need multiple fields together, you pay a small cost in complexity, but for single-field operations (common in analytics), you get 3-4x better cache utilization and enable automatic vectorization by the compiler.

Memory Pools & Custom Allocators

Why Custom Allocators?

- `malloc/new` takes ~20ns on its thread-cache fast path, but occasionally microseconds when it refills from the shared heap or calls `mmap/brk`
- Lock contention in multi-threaded scenarios
- Non-deterministic performance: the tail matters, not the average
- Fragmentation over time

Basic Pool Allocator

This allocator hands out fixed-size slots for objects of type `T` from a pre-allocated pool whose size is fixed at compile time. Allocation and deallocation are a pointer swap on a free list, with zero system calls.

It returns *raw storage*, not constructed objects: construct with placement `new` (or use the `create/destroy` helpers) and destroy before giving the slot back.

```
#include <algorithm>
#include <cstddef>
#include <new>
#include <utility>

template<typename T, size_t PoolSize = 1024>
class PoolAllocator {
    // Each free slot stores the free-list link, so a slot must be big
    // and aligned enough for both a T and a pointer
    union Slot {
        Slot* next;
        alignas(T) std::byte storage[sizeof(T)];
    };

    Slot pool_[PoolSize];
    Slot* free_list_;

public:
    PoolAllocator() {
        for (size_t i = 0; i < PoolSize - 1; ++i) {
            pool_[i].next = &pool_[i + 1];
        }
        pool_[PoolSize - 1].next = nullptr;
        free_list_ = &pool_[0];
    }

    PoolAllocator(const PoolAllocator&) = delete; // free list points into *this
```

```

PoolAllocator& operator=(const PoolAllocator&) = delete;

void* allocate() {
    if (!free_list_) return nullptr; // Pool exhausted
    Slot* slot = free_list_;
    free_list_ = slot->next;
    return slot->storage;
}

void deallocate(void* ptr) {
    Slot* slot = static_cast<Slot*>(ptr);
    slot->next = free_list_;
    free_list_ = slot;
}

template<typename... Args>
T* create(Args&&... args) {
    void* mem = allocate();
    return mem ? new (mem) T(std::forward<Args>(args)...) : nullptr;
}

void destroy(T* obj) {
    obj->~T();
    deallocate(obj);
}
};

```

Note this is not a standard-library Allocator (those have `allocate(n)` for `n` objects, `value_type`, rebinding, etc.), so it can't be plugged into `std::vector` directly. For that, use `std::pmr::unsynchronized_pool_resource` or `std::pmr::monotonic_buffer_resource` with `std::pmr::vector`.

Thread-Local Pools

```

// Avoid contention by giving each thread its own pool
thread_local PoolAllocator<Order, 10000> order_pool;

void process_order() {
    Order* order = order_pool.create(/* ctor args */);
    // ... use order ...
    order_pool.destroy(order);
}

```

Key benefit: Each thread has its own pool, eliminating lock contention entirely.

Caveats: - An object must be freed by the thread that allocated it. If orders cross threads (e.g. through a queue), send them back to the owner for freeing, or use a pool designed for cross-thread frees. - The pool lives in thread-local storage, so $10,000 \times \text{sizeof}(\text{Order})$ bytes are reserved for *every* thread that touches it. For large pools, allocate the slots on the heap once at thread start instead.

Arena Allocator (Bump Allocator)

A bump allocator is the fastest possible allocator - it simply increments a pointer for each allocation. There's no per-object deallocation; instead, you reset the entire arena at once. This is ideal for temporary allocations within a single operation or frame where all objects have the same lifetime.

Warning: This simple implementation doesn't track object types or call destructors. It's only safe for trivially destructible types (POD types, or types where destructor calls aren't critical). For types requiring

cleanup, use a more sophisticated arena that tracks allocated objects.

```
// Super fast: just increment a pointer
// ONLY SAFE FOR TRIVIALY DESTRUCTIBLE TYPES
class ArenaAllocator {
    char* buffer_;
    size_t size_;
    size_t offset_ = 0;

public:
    explicit ArenaAllocator(size_t sz)
        // aligned_alloc requires the size to be a multiple of the alignment
        : size_((sz + 63) & ~size_t{63}) {
        buffer_ = static_cast<char*>(std::aligned_alloc(64, size_));
        if (!buffer_) throw std::bad_alloc();
    }

    ArenaAllocator(const ArenaAllocator&) = delete; // would double-free
    ArenaAllocator& operator=(const ArenaAllocator&) = delete;

    // alignment must be a power of two
    void* allocate(size_t n, size_t alignment = alignof(std::max_align_t)) {
        size_t aligned = (offset_ + alignment - 1) & ~(alignment - 1);

        if (aligned + n > size_) return nullptr;

        offset_ = aligned + n;
        return buffer_ + aligned;
    }

    void reset() {
        offset_ = 0; // Fast reset - no destructor calls!
        // WARNING: Only safe for trivially destructible types
    }

    ~ArenaAllocator() { std::free(buffer_); }
};

// For types requiring destructors, record a destructor per object:
class TypedArenaAllocator {
    struct Destructor {
        void (*fn)(void*);
        void* ptr;
    };

    ArenaAllocator arena_;
    std::vector<Destructor> destructors_;

public:
    TypedArenaAllocator(size_t bytes, size_t max_objects) : arena_(bytes) {
        destructors_.reserve(max_objects); // don't allocate on the hot path
    }

    ~TypedArenaAllocator() { reset(); }
};
```

```

template<typename T, typename... Args>
T* create(Args&&... args) {
    void* mem = arena_.allocate(sizeof(T), alignof(T));
    if (!mem) return nullptr;

    T* obj = new (mem) T(std::forward<Args>(args)...);

    // Track destructor if needed
    if constexpr (!std::is_trivially_destructible_v<T>) {
        destructors_.push_back({
            [](void* p) { static_cast<T*>(p)->~T(); },
            obj
        });
    }

    return obj;
}

void reset() {
    // Call destructors in reverse order of construction
    for (auto it = destructors_.rbegin(); it != destructors_.rend(); ++it) {
        it->fn(it->ptr);
    }
    destructors_.clear();
    arena_.reset();
}
};

```

`std::pmr::monotonic_buffer_resource` is the standard-library version of this idea and works with all `std::pmr` containers.

Compiler Optimizations

Branch Prediction Hints

Modern CPUs use pipelining to execute multiple instructions simultaneously. When the CPU encounters a branch (if/else), it must guess which path to take to keep the pipeline full. A correct prediction means the CPU continues executing at full speed. A misprediction forces the CPU to flush the pipeline and restart - costing 10-20 cycles (~3-7 nanoseconds). In tight loops processing millions of items, mispredictions can cut performance in half.

```

// Help the compiler/CPU predict branches
#define LIKELY(x)    __builtin_expect(!!(x), 1)
#define UNLIKELY(x) __builtin_expect(!!(x), 0)

if (LIKELY(price > 0)) { // Hot path
    process_order();
}

if (UNLIKELY(error_occurred)) { // Cold path
    handle_error();
}

```

```
// C++20: the attribute goes on the branch, after the condition
if (price > 0) [[likely]] { ... }
if (error) [[unlikely]] { ... }
// Not `[[likely]] if (...)`: that marks the whole if-statement as the
// likely path of some enclosing branch, not its own condition.
```

These hints change code layout (hot path falls through, cold path moves out of line). They do not program the CPU's branch predictor, which learns from history; they matter most for branches that are rarely taken but must be fast when they are.

Inlining

A direct call itself is cheap (arguments go in registers on x86-64, and call/return are well predicted). The bigger cost is that the compiler can't optimize across the call boundary: it must assume the callee may read or write any escaped memory, and can't constant-fold, vectorize or hoist through it. Inlining replaces the call with the function body, which removes the call and, more importantly, enables those optimizations. However, aggressive inlining increases code size, which can hurt instruction cache performance.

```
// Force inline for critical paths
__attribute__((always_inline))
inline double calculate_pnl(double price, int qty) {
    return price * qty;
}

// Prevent inlining for cold paths (reduce code size)
__attribute__((noinline))
void handle_rare_error() { ... }
```

Prefetching

Modern CPUs fetch data from memory before you actually need it (hardware prefetcher), but they can only predict sequential or simple stride patterns. For irregular access patterns (like following pointers or indexing with random values), you'll get cache misses. Manual prefetching tells the CPU to load data into cache ahead of time, hiding the memory latency (~50-100ns) behind useful computation. Use prefetch when you know you'll need data soon but the hardware prefetcher won't predict it.

```
// Tell CPU to load data into cache before you need it
void process_orders(const std::vector<Order*>& orders) {
    // Prefetch far enough ahead that the ~100ns memory latency is covered
    // by the work on the orders in between; one element ahead is usually
    // too late. Tune the distance by measurement.
    constexpr size_t PREFETCH_DISTANCE = 8;
    for (size_t i = 0; i < orders.size(); ++i) {
        if (i + PREFETCH_DISTANCE < orders.size()) {
            __builtin_prefetch(orders[i + PREFETCH_DISTANCE], 0 /*read*/, 3 /*keep in all cache levels*/)
        }
        process(orders[i]);
    }
}
```

Interview Questions & Answers

1. “Explain the difference between L1, L2, and L3 cache and their latencies.”

Answer:

Modern CPUs have a hierarchy of caches to bridge the speed gap between the CPU and main memory:

L1 Cache (Level 1): - Smallest and fastest cache, typically 32-64 KB per core - Split into L1i (instruction cache) and L1d (data cache) - Latency: ~4 cycles or about 1-2 nanoseconds - Directly attached to each CPU core - In a trading system, your hottest code and data should fit here

L2 Cache: - Larger, typically 256 KB - 1 MB per core - Unified cache (holds both instructions and data) - Latency: ~12-20 cycles or about 3-7 nanoseconds - Still private to each core in most architectures - Good for frequently accessed data structures that don't fit in L1

L3 Cache (LLC - Last Level Cache): - Largest, typically 8-40 MB shared across all cores - Latency: ~40-75 cycles or about 10-20 nanoseconds - Shared among all cores on the socket - Acts as a victim cache and enables inter-core communication - Important for shared data structures accessed by multiple threads

Main Memory (DRAM): - Latency: ~200-300+ cycles or about 60-100+ nanoseconds - Roughly 50-100x slower than L1 - This is why cache optimization is critical in low-latency systems

Practical Impact:

```
// Example: following 1 million pointers to random Order objects
// Working set fits in L1/L2: ~5-15 cycles per access
// Working set far larger than L3: ~300 cycles per access (a cache miss each)
// Sequential scans are different: the hardware prefetcher streams them,
// so even data in RAM is read at memory bandwidth, not miss latency.
// Random access patterns are what pay the full DRAM latency.
```

Follow-up points to mention: - Cache line size is typically 64 bytes - When you access 1 byte, you load 64 bytes - This is why data layout and alignment matter so much

2. “What is false sharing and how do you avoid it?”

Answer:

False sharing occurs when multiple threads access different variables that happen to reside on the same cache line, causing unnecessary cache coherency traffic and severe performance degradation.

Why it happens: When Thread A writes to variable X and Thread B writes to variable Y, if X and Y are on the same 64-byte cache line, the entire cache line must be invalidated and transferred between CPU cores, even though the threads aren't actually sharing data.

Performance impact: False sharing can degrade performance by 10-100x compared to properly padded structures.

3. “Why would you avoid using `std::vector` in a low-latency system?”

Answer:

While `std::vector` is generally excellent, it has several properties that make it problematic in latency-critical paths:

Problem 1: Dynamic Reallocation

```
// The problem
std::vector<Order> orders;
for (int i = 0; i < 1000; ++i) {
    orders.push_back(order); // Might reallocate!
    // Reallocation = malloc + memcpy + free
    // Can take microseconds, unpredictable
}
```

```

}

// Better: Pre-allocate
std::vector<Order> orders;
orders.reserve(1000); // Do this once at startup
for (int i = 0; i < 1000; ++i) {
    orders.push_back(order); // No reallocation
}

// Best: Fixed size array
std::array<Order, 1000> orders; // Order must be default-constructible
size_t count = 0;
orders[count++] = order; // No allocation ever

```

Problem 2: Heap Allocation

Every allocation goes to the heap allocator which: - Takes locks in multi-threaded programs - Has unpredictable latency (depends on fragmentation) - Might trigger system calls (brk/mmap)

```

// Latency spike example
void process_tick(const Tick& tick) {
    std::vector<Trade> trades; // Allocates on heap
    // ... process ...
} // Deallocates

// This causes allocation/deallocation on every call!

```

Not a problem: iteration speed

Once allocated, a `std::vector`'s elements are contiguous, exactly like a `std::array`'s. Iterating either costs the same: one indirection to find the buffer (hoisted out of the loop), then a sequential scan the prefetcher handles well. Where `std::array` wins is that it can live inside another object or on the stack, avoiding a separate heap block.

Problem 3: Reallocation may copy instead of move

When a vector grows, it moves elements to the new buffer only if the element type's move constructor is `noexcept` (it uses `std::move_if_noexcept` to keep the strong exception guarantee). Otherwise it copies every element:

```

struct Order {
    std::string client_id;
    Order(Order&&) noexcept = default; // without noexcept, growth copies
    // ...
};

```

What to use instead:

1. `std::array` for fixed-size collections:

```

std::array<Order, MAX_ORDERS> order_buffer;
size_t order_count = 0;

```

2. Custom ring buffers:

```

template<typename T, size_t N>
class RingBuffer {
    static_assert((N & (N - 1)) == 0, "N must be a power of 2");
    std::array<T, N> buffer_;
    size_t head_ = 0;
};

```

```

    size_t tail_ = 0;

public:
    bool push(const T& item) {
        size_t next = (tail_ + 1) & (N - 1); // mask instead of % (no division)
        if (next == head_) return false; // Full (holds at most N - 1 items)
        buffer_[tail_] = item;
        tail_ = next;
        return true;
    }
};

```

3. Memory pools with custom allocators:

```

// Pre-allocated pool
template<typename T>
class PoolVector {
    T* data_;
    size_t capacity_;
    size_t size_;

public:
    PoolVector(T* pool, size_t capacity)
        : data_(pool), capacity_(capacity), size_(0) {}

    void push_back(const T& item) {
        assert(size_ < capacity_);
        data_[size_++] = item;
    }
};

```

4. Small vectors with inline storage:

boost::container::small_vector<T, N>, folly::small_vector and llvm::SmallVector keep the first N elements inside the object and only touch the heap beyond that. Use one of these rather than writing your own; getting element lifetimes right in the inline buffer is fiddly.

When std::vector IS acceptable:

- During initialization/setup phase (not in hot path)
- With proper reserve() upfront
- For cold data structures
- With a polymorphic allocator backed by a pre-allocated buffer: std::pmr::vector<T> with a std::pmr::monotonic_buffer_resource

Key interview points: - The issue isn't std::vector itself, but unpredictable allocations - Pre-allocation with reserve() solves most problems - In truly latency-critical code, prefer stack allocation - Always measure - sometimes the convenience is worth the cost

Practical Exercise

Try to optimize this code:

```

// Slow version
std::vector<double> calculate_prices(const std::vector<Trade>& trades) {
    std::vector<double> prices;
    for (const auto& trade : trades) {

```

```

        prices.push_back(trade.price * 1.1);
    }
    return prices;
}

```

What are the issues? 1. No `reserve()` on `prices` - about $\log_2(n)$ reallocations, each copying everything so far 2. A fresh heap allocation (and free, in the caller) on every call 3. `push_back` checks capacity on every iteration, which can block vectorization

The return is *not* a problem: NRVO or, failing that, an implicit move applies, so the vector is never copied.

Optimized version:

```

// Caller owns `prices` and reuses it across calls: after the first call,
// resize() finds enough capacity and allocates nothing
void calculate_prices(const std::vector<Trade>& trades,
                     std::vector<double>& prices) {
    prices.resize(trades.size());
    for (size_t i = 0; i < trades.size(); ++i) {
        prices[i] = trades[i].price * 1.1;
    }
}

```

With `Trade` as a struct of several fields, this loop reads prices at a stride; storing prices in their own array (structure of arrays, above) lets it vectorize cleanly.

Contents

Chapter 2: Concurrency & Threading	1
Table of Contents	1
Thread Management Basics	2
Creating and Managing Threads	2
Hardware Concurrency	2
Memory Ordering Deep Dive	3
Memory Ordering Hierarchy	3
Acquire-Release Semantics Example	3
Sequential Consistency	4
Fences (Memory Barriers)	5
Synchronization Primitives	5
Spin Lock vs Mutex	5
Read-Write Lock (Shared Mutex)	6
Seqlock (Optimistic Read)	7
Lock-Free Data Structures	8
Atomic Operations Fundamentals	8
Lock-Free Single Producer Single Consumer Queue (SPSC)	9
Lock-Free Stack	10
Lock-Free Hash Map (Simplified)	11
MPMC Queue (Multiple Producer Multiple Consumer)	12
SPMC Queues (Single Producer, Multiple Consumers)	14
Thread Pinning & CPU Affinity	19
Why Pin Threads?	19
Linux: Setting CPU Affinity	19
NUMA Awareness	20
Setting Thread Priority	21
Avoiding Context Switches	21
Busy-Wait (Spin) vs Sleep	22
CPU Pause Instruction	22
Interview Questions & Answers	23
1. “What’s the difference between a spin lock and a mutex? When would you use each?” . . .	23
2. “Explain the ABA problem in lock-free programming. How do you solve it?”	23
3. “When would you use memory_order_relaxed vs memory_order_acquire?”	28
4. “How would you implement a lock-free queue?”	30
5. “How would you implement thread pinning for a low-latency trading system?”	31
6. “How do you debug race conditions and data races in concurrent code?”	32
7. “Explain how you would implement a wait-free queue and why it matters.”	34
Practical Examples	36
Complete Low-Latency Pipeline	36

Chapter 2: Concurrency & Threading

Table of Contents

1. Thread Management Basics
2. Memory Ordering Deep Dive
3. Synchronization Primitives
4. Lock-Free Data Structures
5. Thread Pinning & CPU Affinity
6. Avoiding Context Switches
7. Interview Questions & Answers

Thread Management Basics

Proper thread management is the foundation of concurrent programming. In low-latency systems, threads are typically pinned to specific CPU cores and run continuously without blocking. Understanding thread creation, lifecycle, and hardware capabilities is essential for building performant multi-threaded applications.

Creating and Managing Threads

```
#include <thread>
#include <pthread.h>

// Basic thread creation
std::thread worker([]() {
    // Worker logic
});

// Join or detach
worker.join();    // Wait for completion
// OR
worker.detach();  // Run independently
```

join() vs detach(): - **join():** Blocks the calling thread until the worker thread completes. The parent thread waits and can safely access any results. Use this when you need to ensure the work is done before proceeding, or when the worker accesses local variables that would go out of scope. - **detach():** The worker thread runs independently and cleans up automatically when it finishes. The parent thread continues immediately without waiting. Use this for fire-and-forget tasks like logging or background cleanup. Warning: detached threads can't be joined later, and the program may exit before they finish. - **Critical: Must call one or the other!** If a `std::thread` object goes out of scope while still joinable (neither joined nor detached), its destructor calls `std::terminate()` and aborts the program. This is a safety mechanism to prevent accidental thread leaks and undefined behavior. C++20's `std::jthread` joins automatically in its destructor and supports cooperative cancellation via `std::stop_token`.

```
void dangerous() {
    std::thread worker([]() { /* work */ });
    // Forgot to join() or detach()
} // ERROR: std::terminate() called, program aborts!

void safe_join() {
    std::thread worker([]() { /* work */ });
    worker.join(); // OK: waited for completion
}

void safe_detach() {
    std::thread worker([]() { /* work */ });
    worker.detach(); // OK: runs independently
}
```

Hardware Concurrency

Understanding your system's hardware capabilities is crucial for optimal thread allocation. Creating more threads than available hardware threads leads to context switching overhead, while underutilizing cores wastes potential parallelism. In low-latency systems with CPU-bound work (no blocking I/O), you typically create one thread per physical core for maximum performance. For I/O-bound workloads where threads block waiting for network or disk, you may need more threads to keep cores busy while others are blocked.

```

// Number of *logical* CPUs (hyperthreads count separately), or 0 if unknown
unsigned int num_cpus = std::thread::hardware_concurrency();

std::vector<std::thread> workers;
workers.reserve(num_cpus);

for (unsigned i = 0; i < num_cpus; ++i) {
    workers.emplace_back(worker_function, i);
}

```

`hardware_concurrency()` counts logical CPUs, so on a machine with SMT it returns twice the number of physical cores. To get one thread per physical core, read the topology (`lscpu -e`, `/sys/devices/system/cpu/cpu*/topology/thread_siblings_list`, or `hwloc`). In practice, latency-critical threads are placed on specific, isolated cores chosen by configuration, not by a loop over all CPUs.

Memory Ordering Deep Dive

Memory ordering controls how atomic operations synchronize between threads. Modern CPUs and compilers reorder instructions for performance, but this can break multi-threaded code if not managed carefully. C++ provides different memory ordering guarantees - from **relaxed** (no synchronization, just atomicity) to **seq_cst** (full sequential consistency). Choosing the right memory order is critical: too weak and your code has data races; too strong and you pay unnecessary performance costs. Understanding memory ordering is essential for writing correct and efficient lock-free code.

Memory Ordering Hierarchy

```

// Roughly from weakest to strongest:
std::memory_order_relaxed    // No synchronization, just atomicity
std::memory_order_consume    // Discouraged: every compiler treats it as acquire
std::memory_order_acquire    // For loads: later accesses can't move before it
std::memory_order_release    // For stores: earlier accesses can't move after it
std::memory_order_acq_rel    // For read-modify-writes: both
std::memory_order_seq_cst    // Acquire/release plus one total order (the default)

```

Acquire-Release Semantics Example

Acquire-release is the most commonly used memory ordering pair in lock-free programming. The producer uses `memory_order_release` on a store to guarantee that all its previous writes are visible when another thread performs an `acquire` load that sees that value. The consumer uses `memory_order_acquire` on a load to ensure it sees all writes that happened before the corresponding release. This creates a synchronization point: everything before the release happens-before everything after the acquire. This is cheaper than full sequential consistency but provides the synchronization needed for correct communication between threads.

```

// One-shot hand-off: publish() may be called once. Calling it again while
// a consumer is reading would be a data race on data_ (use a queue or a
// seqlock for repeated updates).
class DataChannel {
    int data_[100];
    std::atomic<int> count_{0};

public:
    // Producer
    void publish(const int* values, int n) { // n <= 100

```

```

    // Write data first
    for (int i = 0; i < n; ++i) {
        data_[i] = values[i];
    }

    // Release: All writes above are visible to acquire
    count_.store(n, std::memory_order_release);
}

// Consumer
bool consume(int* out, int max_count) {
    // Acquire: See all writes before the release
    int n = count_.load(std::memory_order_acquire);

    if (n == 0) return false;

    int to_copy = std::min(n, max_count);
    for (int i = 0; i < to_copy; ++i) {
        out[i] = data_[i]; // Safe to read
    }

    return true;
}
};

```

Sequential Consistency

Sequential consistency (`memory_order_seq_cst`) is the strongest and most intuitive memory ordering - it guarantees that all threads see all sequentially consistent operations in the same global order, and that this order respects the program order within each thread. In other words, operations appear to execute in the order they're written in your code, and all threads agree on that order.

The extra cost is concentrated in one place: preventing a store from being reordered with a *later load* (StoreLoad), which even x86 allows. On x86, `seq_cst` loads are plain `movs` (same as `acquire`), read-modify-writes cost the same at every ordering (they're all lock-prefixed), and only `seq_cst stores` pay extra (`xchg` or `mov + mfence`, ~20-30 cycles).

```

// Global order of all seq_cst operations
std::atomic<int> x{0}, y{0};

// Thread 1
void thread1() {
    x.store(1, std::memory_order_seq_cst);
    int r1 = y.load(std::memory_order_seq_cst);
}

// Thread 2
void thread2() {
    y.store(1, std::memory_order_seq_cst);
    int r2 = x.load(std::memory_order_seq_cst);
}

// With seq_cst: Impossible for r1 == 0 && r2 == 0
// With relaxed: Could both be 0!

```

Fences (Memory Barriers)

Fences (also called memory barriers) move the ordering off the atomic operation and onto a separate statement. The pairing rule: if thread A does a **release fence** and then an atomic store (even relaxed), and thread B does an atomic load (even relaxed) that reads that value and then an **acquire fence**, then everything A wrote before its fence is visible to B after its fence. This is useful when one fence can cover several relaxed atomics, or when the consumer only needs ordering on the path where the flag was actually seen. Fences are harder to reason about than ordered operations, so use them deliberately.

```
// Alternative to acquire/release on atomics
std::atomic<bool> ready{false};
int data = 0;

// Producer
void produce() {
    data = 42;
    std::atomic_thread_fence(std::memory_order_release);
    ready.store(true, std::memory_order_relaxed); // Can be relaxed - fence does the work
}

// Consumer
void consume() {
    while (!ready.load(std::memory_order_relaxed)) { // Can be relaxed
        // spin
    }
    std::atomic_thread_fence(std::memory_order_acquire);
    assert(data == 42); // Guaranteed
}
```

Synchronization Primitives

Synchronization primitives are the building blocks for coordinating access to shared data between threads. While lock-free techniques avoid blocking entirely, traditional synchronization primitives use locks to ensure mutual exclusion. Each primitive has different performance characteristics and use cases:

- **Mutex** (`std::mutex`) - Basic mutual exclusion, blocks waiting threads
- **Spin Lock** - Busy-waits instead of sleeping, fastest for very short critical sections
- **Read-Write Lock** (`std::shared_mutex`) - Multiple readers or single writer
- **Semaphore** (`std::counting_semaphore`) - Controls access to a resource pool
- **Condition Variable** (`std::condition_variable`) - Waits for a condition, used with mutex
- **Seqlock** - Optimistic lock for readers, writers block briefly
- **Lock Guard** (`std::lock_guard`, `std::unique_lock`, `std::scoped_lock`) - RAII wrappers that automatically acquire/release locks, preventing deadlocks and resource leaks

In low-latency systems, spinlocks and seqlocks are preferred over mutexes when critical sections are short, as they avoid expensive context switches.

Spin Lock vs Mutex

The fundamental trade-off: spinlocks burn CPU cycles busy-waiting but never sleep, while mutexes sleep when contended and pay for a kernel wake-up. Uncontended, both are a single atomic instruction (~20ns): Linux's `std::mutex` only enters the kernel (futex) when it actually has to wait. The difference is in the *contended* case: for very short critical sections, a spinner acquires the lock within nanoseconds, while a sleeping waiter pays a context switch (~1-10 s) to wake up. For longer critical sections, mutexes are better as they free the CPU for other work.

```

#include <atomic>
#include <immintrin.h> // _mm_pause

inline void cpu_relax() { _mm_pause(); } // x86 PAUSE; see "CPU Pause Instruction"

// Spin Lock: Good for very short critical sections, on dedicated cores
class SpinLock {
    std::atomic<bool> locked_{false};

public:
    void lock() {
        for (;;) {
            // Try to grab it: one read-modify-write
            if (!locked_.exchange(true, std::memory_order_acquire)) {
                return;
            }
            // Wait with plain loads, not repeated exchanges: loads keep the
            // cache line shared instead of bouncing it between waiting cores
            // ("test-and-test-and-set")
            while (locked_.load(std::memory_order_relaxed)) {
                cpu_relax();
            }
        }
    }

    bool try_lock() {
        return !locked_.load(std::memory_order_relaxed) &&
            !locked_.exchange(true, std::memory_order_acquire);
    }

    void unlock() {
        locked_.store(false, std::memory_order_release);
    }
};

// Works with the standard RAII wrappers:
SpinLock spin;
{
    std::lock_guard<SpinLock> lock(spin);
    ++counter; // Very short critical section
}

```

Spinlock hazard: if the thread holding the lock is preempted, every waiter spins uselessly until it is scheduled again, possibly for a whole scheduler time slice (milliseconds). Spinlocks are only safe when lock holders run on dedicated cores that nothing else is scheduled on.

Read-Write Lock (Shared Mutex)

Read-write locks optimize for the common case where data is read frequently but written rarely. Multiple threads can hold read (shared) locks simultaneously since reads don't conflict with each other, maximizing concurrency. Write (exclusive) locks block all other access. This is ideal for data structures like configuration, lookup tables, or caches where reads vastly outnumber writes.

```

#include <shared_mutex>

```

```

class DataStore {
    mutable std::shared_mutex mutex_;
    std::unordered_map<int, std::string> data_;

public:
    // Multiple readers can hold this simultaneously
    std::string read(int key) const {
        std::shared_lock lock(mutex_);
        auto it = data_.find(key);
        return it != data_.end() ? it->second : "";
    }

    // Exclusive write access
    void write(int key, const std::string& value) {
        std::unique_lock lock(mutex_);
        data_[key] = value;
    }
};

```

Seqlock (Optimistic Read)

Seqlocks take read-write locks further by making reads completely lock-free through optimistic concurrency. Readers never block - they speculatively read data and check afterward if a write occurred during the read. If a write happened, readers retry. The writer increments a sequence number before and after writing (making it odd during writes). This is extremely fast for read-heavy workloads with small data (e.g. publishing the latest top-of-book), but readers may retry multiple times under heavy write contention.

Two details make or break a seqlock in C++: - **The data must be read with atomic operations.** Readers deliberately read while the writer may be writing; with plain (non-atomic) fields that is a data race, i.e. UB, even though the torn value would be discarded. Relaxed atomic loads/stores cost the same as plain moves on x86. - **The writer needs a release fence after marking the sequence odd**, so that a reader who sees any new data is guaranteed to also see the odd sequence number and retry (Boehm, “Can Seqlocks Get Along with Programming Language Memory Models?”, 2012).

```

// Single writer, many readers. T must be trivially copyable.
template<typename T>
class SeqLock {
    static_assert(std::is_trivially_copyable_v<T>);
    static constexpr size_t N = (sizeof(T) + 7) / 8;

    std::atomic<uint64_t> seq_{0};
    std::atomic<uint64_t> data_[N] = {}; // payload as relaxed atomic words

public:
    void store(const T& value) { // only ever called from one thread
        uint64_t words[N] = {};
        std::memcpy(words, &value, sizeof(T));

        const uint64_t s = seq_.load(std::memory_order_relaxed);
        seq_.store(s + 1, std::memory_order_relaxed); // odd: writing
        std::atomic_thread_fence(std::memory_order_release); // odd before data
        for (size_t i = 0; i < N; ++i) {
            data_[i].store(words[i], std::memory_order_relaxed);
        }
        seq_.store(s + 2, std::memory_order_release); // even: done
    }
};

```

```

    }

    T load() const {
        uint64_t words[N];
        uint64_t s0, s1;
        do {
            s0 = seq_.load(std::memory_order_acquire);
            for (size_t i = 0; i < N; ++i) {
                words[i] = data_[i].load(std::memory_order_relaxed);
            }
            std::atomic_thread_fence(std::memory_order_acquire); // data before re-check
            s1 = seq_.load(std::memory_order_relaxed);
        } while ((s0 & 1) || s0 != s1); // writer was active: retry

        T value;
        std::memcpy(&value, words, sizeof(T));
        return value;
    }
};

// Usage
struct TopOfBook { int64_t bid_px, ask_px; int32_t bid_qty, ask_qty; };
SeqLock<TopOfBook> tob;
// writer thread: tob.store(new_tob);
// reader threads: TopOfBook snapshot = tob.load();

```

With more than one writer, writers must first serialize among themselves (e.g. a spinlock around `store`).

ThreadSanitizer does not model standalone `atomic_thread_fence` (GCC warns: `-Wtsan`), so it can't fully verify fence-based code like this. Test seqlocks with a stress test that checks every snapshot for consistency (e.g. writer keeps `ask_px == bid_px + 1`; reader asserts it).

Lock-Free Data Structures

Lock-free data structures use atomic operations instead of locks, avoiding context switches and providing better worst-case latency guarantees. This section covers the fundamentals of atomic operations and progressively more complex lock-free structures.

Atomic Operations Fundamentals

Atomic operations are indivisible - they complete fully or not at all, with no observable intermediate state. They're the building blocks of lock-free programming, providing thread-safe operations without locks. Understanding different memory orderings (relaxed, acquire/release, `seq_cst`) is crucial for both correctness and performance.

```

#include <atomic>

// Basic atomic operations
std::atomic<int> counter{0};

// Different memory orderings for different guarantees
// Relaxed: No synchronization, just atomicity
counter.fetch_add(1, std::memory_order_relaxed);

```

```

// Acquire-Release: Synchronization between threads
// Release: All writes before this are visible to acquire
int data = 0;
std::atomic<bool> ready{false};

// Producer thread
data = 42; // Write data first
ready.store(true, std::memory_order_release); // Publish

// Consumer thread
if (ready.load(std::memory_order_acquire)) { // Synchronize
    assert(data == 42); // Guaranteed to see the write
}

// Sequential consistency: Strongest guarantee, most expensive
counter.fetch_add(1, std::memory_order_seq_cst);

// Compare-and-swap (CAS) - fundamental building block
int expected = 10;
int desired = 20;
if (counter.compare_exchange_strong(expected, desired,
                                    std::memory_order_acq_rel, // on success
                                    std::memory_order_acquire)) { // on failure
    // Success: counter was 10, now 20
} else {
    // Failure: expected now contains actual counter value
}

```

Lock-Free Single Producer Single Consumer Queue (SPSC)

The simplest and fastest lock-free queue - each index is only written by one thread, eliminating the need for compare-and-swap operations. The producer owns the tail index, the consumer owns the head index, and they only read each other's indices. This single-ownership model makes SPSC queues extremely efficient and is the most common pattern in low-latency pipelines. This is the reference implementation for the rest of the chapter.

```

template<typename T, size_t Capacity>
class SPSCQueue {
    static_assert((Capacity & (Capacity - 1)) == 0, "Capacity must be a power of 2");
    static constexpr size_t MASK = Capacity - 1;

    // Each index on its own cache line to avoid false sharing
    alignas(64) std::atomic<size_t> head_{0}; // Consumer writes
    alignas(64) std::atomic<size_t> tail_{0}; // Producer writes
    alignas(64) std::array<T, Capacity> buffer_;

public:
    // Producer thread only
    bool try_push(const T& item) {
        const size_t tail = tail_.load(std::memory_order_relaxed); // own index
        const size_t next = (tail + 1) & MASK;

        if (next == head_.load(std::memory_order_acquire)) {
            return false; // Full
        }
    }

```

```

    buffer_[tail] = item;
    tail_.store(next, std::memory_order_release); // publish the item
    return true;
}

// Consumer thread only
bool try_pop(T& item) {
    const size_t head = head_.load(std::memory_order_relaxed); // own index

    if (head == tail_.load(std::memory_order_acquire)) {
        return false; // Empty
    }

    item = buffer_[head];
    head_.store((head + 1) & MASK, std::memory_order_release); // free the slot
    return true;
}
};

```

Why each ordering is needed: - relaxed load of your own index: no other thread writes it - Producer's release store of tail_ + consumer's acquire load of it: the consumer sees the item's contents once it sees the new tail - Consumer's release store of head_ + producer's acquire load of it: the consumer's read of the slot happens-before the producer overwrites that slot. This one is easy to forget; without it the producer can overwrite an item the consumer is still copying - Power-of-2 capacity: masking instead of % (no division); one slot stays empty so that "full" (next == head) is distinguishable from "empty" (tail == head)

Common optimization: each side keeps a private copy of the *other* side's index and only re-reads the shared atomic when the cached value says the queue is full (producer) or empty (consumer). That removes most cross-core cache-line traffic (used by e.g. rigtorp::SPSCQueue).

Lock-Free Stack

A lock-free stack allows any number of threads to push and pop concurrently using compare-and-swap (CAS) operations. This example shows the basic algorithm; **it is not safe as written** because of two related problems: - **Use-after-free:** pop reads old_head->next while another thread may already have popped and deleted old_head. - **ABA:** a pointer can have the same value at different times but refer to different objects, so a CAS can succeed when it shouldn't (see interview question 2).

Production code needs safe memory reclamation (hazard pointers, epochs) and ABA protection (tagged pointers, or the reclamation scheme itself).

```

template<typename T>
class LockFreeStack {
    struct Node {
        T data;
        Node* next;
        Node(const T& d) : data(d), next(nullptr) {}
    };

    std::atomic<Node*> head_{nullptr};

public:
    void push(const T& data) {
        Node* new_node = new Node(data);
        new_node->next = head_.load(std::memory_order_relaxed);
    }
};

```

```

    // CAS loop: retry until successful
    while (!head_.compare_exchange_weak(
        new_node->next,
        new_node,
        std::memory_order_release,
        std::memory_order_relaxed)) {
        // If CAS fails, new_node->next is updated with current head
        // Loop and try again
    }
}

bool pop(T& result) {
    Node* old_head = head_.load(std::memory_order_acquire);

    while (old_head != nullptr) {
        // Try to swing head to next node
        // UNSAFE: old_head may already have been deleted by another pop
        if (head_.compare_exchange_weak(
            old_head,
            old_head->next,
            std::memory_order_release,
            std::memory_order_acquire)) {

            result = old_head->data;
            delete old_head; // In production, use hazard pointers
            return true;
        }
        // If CAS fails, old_head is updated, retry
    }

    return false; // Stack was empty
}
};

```

compare_exchange_weak vs compare_exchange_strong: - **weak:** Can spuriously fail even when the comparison succeeds (returns false; **expected** is rewritten with the same value). This happens on architectures that implement CAS with load-linked/store-conditional pairs (older ARM, POWER), where the store-conditional can fail due to an interrupt or cache-line eviction. It's cheaper there because it's a single LL/SC attempt. On x86 (`lock cmpxchg`) and ARMv8.1+ with LSE (`cas`), weak and strong compile to the same instruction. - **strong:** Never spuriously fails - only fails when values actually differ. On LL/SC architectures it's a loop around the weak version. - **When to use weak:** In loops (like above) where you're retrying anyway. The spurious failure just means one extra iteration. - **When to use strong:** When failure is expensive or you're not in a loop. Example: one-shot initialization.

Lock-Free Hash Map (Simplified)

Lock-free hash maps allow concurrent insertions and lookups without blocking. Each bucket is essentially a lock-free linked list. This simplified version is insert-only (no updates, deletion or resizing, and duplicate keys are allowed), which is also why it needs no memory reclamation: nodes are never freed. Production implementations are significantly more complex and typically use split-ordered lists (Shalev & Shavit) or open addressing with atomic slots.

```

template<typename K, typename V, size_t Size>
class LockFreeHashMap {

```

```

struct Node {
    K key;
    V value;
    std::atomic<Node*> next{nullptr};
};

std::array<std::atomic<Node*>, Size> buckets_{};

size_t hash(const K& key) const {
    return std::hash<K>{}(key) % Size;
}

public:
void insert(const K& key, const V& value) {
    size_t bucket = hash(key);
    Node* new_node = new Node{key, value, nullptr};

    Node* head = buckets_[bucket].load(std::memory_order_acquire);
    new_node->next.store(head, std::memory_order_relaxed);

    while (!buckets_[bucket].compare_exchange_weak(
        head,
        new_node,
        std::memory_order_release,
        std::memory_order_acquire)) {
        new_node->next.store(head, std::memory_order_relaxed);
    }
}

bool find(const K& key, V& value) {
    size_t bucket = hash(key);
    Node* current = buckets_[bucket].load(std::memory_order_acquire);

    while (current != nullptr) {
        if (current->key == key) {
            value = current->value;
            return true;
        }
        current = current->next.load(std::memory_order_acquire);
    }

    return false;
}
};

```

MPMC Queue (Multiple Producer Multiple Consumer)

MPMC queues allow any number of threads to produce and consume simultaneously. This implementation uses sequence numbers to track cell states and coordinate between competing producers and consumers. The algorithm is Dmitry Vyukov's bounded MPMC queue, which uses sequence numbers to avoid ABA problems.

Strictly speaking it is not lock-free: a producer that claims a cell and is then preempted before publishing it blocks consumers at that cell until it resumes. In practice it is very fast and widely used; just don't claim a lock-free progress guarantee for it in an interview.

```

template<typename T, size_t Size>
class MPMCQueue {
    struct Cell {
        std::atomic<size_t> sequence;
        T data;
    };

    static constexpr size_t MASK = Size - 1;
    static_assert((Size & MASK) == 0, "Size must be power of 2");

    alignas(64) std::atomic<size_t> enqueue_pos_{0};
    alignas(64) std::atomic<size_t> dequeue_pos_{0};

    std::array<Cell, Size> buffer_;

public:
    MPMCQueue() {
        for (size_t i = 0; i < Size; ++i) {
            buffer_[i].sequence.store(i, std::memory_order_relaxed);
        }
    }

    bool push(const T& data) {
        Cell* cell;
        size_t pos = enqueue_pos_.load(std::memory_order_relaxed);

        for (;;) {
            cell = &buffer_[pos & MASK];
            size_t seq = cell->sequence.load(std::memory_order_acquire);
            intptr_t diff = (intptr_t)seq - (intptr_t)pos;

            if (diff == 0) {
                // Slot is available
                if (enqueue_pos_.compare_exchange_weak(
                    pos, pos + 1, std::memory_order_relaxed)) {
                    break;
                }
            } else if (diff < 0) {
                // Queue is full
                return false;
            } else {
                // Another producer got ahead, update position
                pos = enqueue_pos_.load(std::memory_order_relaxed);
            }
        }

        cell->data = data;
        cell->sequence.store(pos + 1, std::memory_order_release);
        return true;
    }

    bool pop(T& data) {
        Cell* cell;
        size_t pos = dequeue_pos_.load(std::memory_order_relaxed);

```

```

for (;;) {
    cell = &buffer_[pos & MASK];
    size_t seq = cell->sequence.load(std::memory_order_acquire);
    intptr_t diff = (intptr_t)seq - (intptr_t)(pos + 1);

    if (diff == 0) {
        // Data is available
        if (dequeue_pos_.compare_exchange_weak(
            pos, pos + 1, std::memory_order_relaxed)) {
            break;
        }
    } else if (diff < 0) {
        // Queue is empty
        return false;
    } else {
        // Another consumer got ahead
        pos = dequeue_pos_.load(std::memory_order_relaxed);
    }
}

data = cell->data;
cell->sequence.store(pos + MASK + 1, std::memory_order_release);
return true;
}
};

```

How the algorithm works:

The key insight is using sequence numbers to track the state of each cell. Each cell has a sequence number that encodes whether it's empty, being written, or contains data. Producers and consumers maintain separate position counters and compete using CAS to claim cells.

When a producer wants to enqueue, it reads the current `enqueue_pos` and checks the corresponding cell's sequence number. If `seq == pos`, the cell is empty and available. The producer uses CAS to claim the position by incrementing `enqueue_pos`. If successful, it writes data and updates the sequence to signal data is ready (`pos + 1`). If `seq < pos`, the cell is still occupied (queue full); if `seq > pos`, another producer got ahead.

Consumers work symmetrically: they look for cells where `seq == pos + 1` (data ready). After reading, they update the sequence to `pos + MASK + 1`, effectively marking the cell as empty for the next cycle around the ring buffer. The MASK ensures proper wrapping.

The beauty is that sequence numbers prevent ABA problems - even if we wrap around the ring buffer, the sequence numbers distinguish between "empty on first pass" and "empty after wrapping." Producers and consumers only retry their CAS operations when there's actual contention (with the caveat about a preempted producer above).

SPMC Queues (Single Producer, Multiple Consumers)

"One producer, several consumers" means two different things, and the designs are different:

	Work distribution	Broadcast (fan-out)
Each item is seen by	exactly one consumer	every consumer
Typical use	a pool of workers processing orders or requests	one market data thread feeding several strategies

	Work distribution	Broadcast (fan-out)
Consumers coordinate with	each other (they compete for items)	nobody; each keeps its own position

Work Distribution With one producer, the producer side needs no CAS; only consumers compete, with a CAS on the shared read position. This is Vyukov's MPMC queue from above with the producer simplified:

```
template<typename T, size_t Capacity>
class SPMCQueue {
    static_assert((Capacity & (Capacity - 1)) == 0, "Capacity must be a power of 2");
    static constexpr size_t MASK = Capacity - 1;

    struct alignas(64) Cell {                // one cell per cache line
        std::atomic<size_t> sequence;
        T data;
    };

    alignas(64) size_t tail_ = 0;            // producer only: not atomic
    alignas(64) std::atomic<size_t> head_{0}; // consumers compete on this
    std::array<Cell, Capacity> buffer_;

public:
    SPMCQueue() {
        for (size_t i = 0; i < Capacity; ++i) {
            buffer_[i].sequence.store(i, std::memory_order_relaxed);
        }
    }

    // Producer thread only
    bool try_push(const T& item) {
        Cell& cell = buffer_[tail_ & MASK];
        // sequence == tail_: a consumer has released this cell for this lap
        if (cell.sequence.load(std::memory_order_acquire) != tail_) {
            return false; // Full (or the consumer of the previous lap isn't done)
        }
        cell.data = item;
        cell.sequence.store(tail_ + 1, std::memory_order_release); // publish
        ++tail_;
        return true;
    }

    // Any number of consumer threads
    bool try_pop(T& item) {
        size_t pos = head_.load(std::memory_order_relaxed);
        for (;;) {
            Cell& cell = buffer_[pos & MASK];
            const size_t seq = cell.sequence.load(std::memory_order_acquire);
            const intptr_t diff = static_cast<intptr_t>(seq) - static_cast<intptr_t>(pos + 1);

            if (diff == 0) {
                // Item published: try to claim it
                if (head_.compare_exchange_weak(pos, pos + 1, std::memory_order_relaxed)) {
                    item = cell.data;
                }
            }
        }
    }
};
```

```

        // Release the cell for the producer's next lap
        cell.sequence.store(pos + Capacity, std::memory_order_release);
        return true;
    }
    // CAS failed: pos now holds the current head, retry
} else if (diff < 0) {
    return false; // Empty
} else {
    pos = head_.load(std::memory_order_relaxed); // another consumer got ahead
}
}
};

```

As with the MPMC queue, a consumer preempted between winning the CAS and releasing the cell stalls the producer when it laps around to that cell.

The usual alternative: one SPSC queue per consumer, with the producer choosing the queue (round-robin, or by key such as instrument, so all updates for one instrument go to the same worker and stay in order). There are no CAS operations and no shared write location between consumers, but also no automatic load balancing: a slow consumer's queue fills while others sit idle. In trading systems partitioning by key is usually preferred, because per-instrument ordering matters.

Broadcast, Producer Waits for the Slowest (Disruptor-style) Every consumer reads every item, so a slot can only be reused once *all* consumers have passed it. The producer publishes a sequence number; each consumer publishes how far it has read, on its own cache line. This is the core of the LMAX Disruptor:

```

template<typename T, size_t Capacity, size_t MaxConsumers>
class BroadcastQueue {
    static_assert((Capacity & (Capacity - 1)) == 0, "Capacity must be a power of 2");
    static constexpr size_t MASK = Capacity - 1;

    struct alignas(64) Cursor {
        std::atomic<uint64_t> next{0}; // next sequence this consumer will read
    };

    alignas(64) std::atomic<uint64_t> published_{0}; // number of items published
    alignas(64) uint64_t cached_min_ = 0; // producer only
    std::array<Cursor, MaxConsumers> consumers_;
    size_t num_consumers_;
    std::array<T, Capacity> buffer_;

    // 64-bit sequence numbers never wrap in practice (centuries at 1e9/s),
    // so "full" and "empty" are simple subtractions

public:
    explicit BroadcastQueue(size_t num_consumers) : num_consumers_(num_consumers) {}

    // Producer thread only
    bool try_publish(const T& item) {
        const uint64_t seq = published_.load(std::memory_order_relaxed);
        if (seq - cached_min_ >= Capacity) {
            // Looks full: re-read the slowest consumer's position. Caching it
            // means the producer usually touches no consumer cache line.
            uint64_t min_next = seq;

```

```

        for (size_t c = 0; c < num_consumers_; ++c) {
            min_next = std::min(min_next,
                                consumers_[c].next.load(std::memory_order_acquire));
        }
        cached_min_ = min_next;
        if (seq - cached_min_ >= Capacity) {
            return false; // The slowest consumer is a full ring behind
        }
    }
    buffer_[seq & MASK] = item;
    published_.store(seq + 1, std::memory_order_release);
    return true;
}

// Consumer `c` only (each consumer index used by exactly one thread)
bool try_consume(size_t c, T& item) {
    std::atomic<uint64_t>& cursor = consumers_[c].next;
    const uint64_t next = cursor.load(std::memory_order_relaxed);
    if (next == published_.load(std::memory_order_acquire)) {
        return false; // Nothing new
    }
    item = buffer_[next & MASK];
    cursor.store(next + 1, std::memory_order_release); // done with this slot
    return true;
}
};

```

Each producer/consumer pair has the same orderings as the SPSC queue: `published_` is release/acquire so a consumer sees the item, and each consumer's cursor is release/acquire so the producer never overwrites a slot that consumer is still reading.

Properties: - No CAS anywhere; each consumer only writes its own cursor. - **Back-pressure:** one stalled consumer eventually blocks the producer for everyone. That's right when no consumer may miss data (e.g. a risk engine, a journal), and wrong when the producer must never wait. - Consumers are registered up front (MaxConsumers), because the producer has to read their cursors. - **Batching:** a consumer that is behind can read `published_` once and process everything up to it before storing its cursor once, which amortizes the cache-line transfers. The real Disruptor does this, and also lets consumers depend on each other (e.g. "the journaller must finish a slot before the business logic sees it").

Broadcast, Producer Never Waits (Seqlock Ring) For market data, a slow consumer must not delay the producer, or every other consumer: it's better for the slow one to notice it fell behind and resynchronize. Here each slot is a small seqlock (Seqlock above). The producer overwrites slots unconditionally; a consumer checks the slot's sequence number to tell "not written yet" from "already overwritten":

```

// Single producer, any number of consumers, which can come and go freely:
// the producer doesn't know they exist. T must be trivially copyable.
template<typename T, size_t Capacity>
class OverwriteRing {
    static_assert((Capacity & (Capacity - 1)) == 0, "Capacity must be a power of 2");
    static_assert(std::is_trivially_copyable_v<T>);
    static constexpr size_t MASK = Capacity - 1;
    static constexpr size_t WORDS = (sizeof(T) + 7) / 8;

    struct alignas(64) Slot {
        // Message n is written as 2n+1 (in progress), then 2n+2 (complete)

```

```

        std::atomic<uint64_t> seq{0};
        std::atomic<uint64_t> data[WORDS] = {}; // payload as relaxed atomic words
    };

    alignas(64) std::atomic<uint64_t> published_{0}; // messages published so far
    alignas(64) uint64_t next_ = 0; // producer only
    std::array<Slot, Capacity> slots_;

public:
    enum class Result { Ok, Empty, Overrun };

    // Producer thread only: never blocks, never fails
    void publish(const T& value) {
        uint64_t words[WORDS] = {};
        std::memcpy(words, &value, sizeof(T));

        const uint64_t n = next_++;
        Slot& slot = slots_[n & MASK];
        slot.seq.store(2 * n + 1, std::memory_order_relaxed); // writing
        std::atomic_thread_fence(std::memory_order_release);
        for (size_t i = 0; i < WORDS; ++i) {
            slot.data[i].store(words[i], std::memory_order_relaxed);
        }
        slot.seq.store(2 * n + 2, std::memory_order_release); // complete
        published_.store(n + 1, std::memory_order_release);
    }

    // Consumer: `next` is this consumer's own position (message number)
    Result try_read(uint64_t& next, T& out) const {
        const Slot& slot = slots_[next & MASK];
        const uint64_t expected = 2 * next + 2;

        const uint64_t s0 = slot.seq.load(std::memory_order_acquire);
        if (s0 < expected) return Result::Empty; // not written (or being written)
        if (s0 > expected) return Result::Overrun; // producer lapped us

        uint64_t words[WORDS];
        for (size_t i = 0; i < WORDS; ++i) {
            words[i] = slot.data[i].load(std::memory_order_relaxed);
        }
        std::atomic_thread_fence(std::memory_order_acquire);
        if (slot.seq.load(std::memory_order_relaxed) != s0) {
            return Result::Overrun; // overwritten while we read it
        }

        std::memcpy(&out, words, sizeof(T));
        ++next;
        return Result::Ok;
    }

    // Where a new or lapped consumer should start: the next message to be written
    uint64_t resync_position() const {
        return published_.load(std::memory_order_acquire);
    }
}

```

```

};

// Consumer loop
void consume(const OverwriteRing<Quote, 4096>& ring) {
    uint64_t next = ring.resync_position(); // join at the live edge
    Quote q;
    for (;;) {
        switch (ring.try_read(next, q)) {
            case OverwriteRing<Quote, 4096>::Result::Ok:
                on_quote(q);
                break;
            case OverwriteRing<Quote, 4096>::Result::Empty:
                cpu_relax();
                break;
            case OverwriteRing<Quote, 4096>::Result::Overrun:
                // We missed messages: the book is now wrong. Rebuild it from
                // a snapshot, then continue from the live edge.
                request_snapshot();
                next = ring.resync_position();
                break;
        }
    }
}

```

Properties: - The producer's cost doesn't depend on the number of consumers, and it never reads anything consumers write, so there is no cache-line traffic from consumers back to the producer. - Consumers are read-only, so they can attach and detach at any time, and the ring can live in shared memory read by other *processes* (a common way to distribute market data on one host). - A consumer that falls behind by more than **Capacity** messages loses data, but it always knows: it sees **Overrun** instead of silently reading a newer message. Recovery is the same as for a network gap (snapshot, then resume). - The seqlock rules from above apply: payload read and written as relaxed atomics, a release fence after marking the slot odd, and an acquire fence before re-checking.

Choosing between the broadcast designs: use the Disruptor-style queue when every consumer must see every message (journaling, risk, order state), and the overwrite ring when the producer must never slow down and consumers can recover from a gap (market data distribution).

Thread Pinning & CPU Affinity

Why Pin Threads?

Thread pinning (also called CPU affinity) binds a thread to specific CPU cores, preventing the OS scheduler from moving it around. This has several critical benefits for low-latency systems:

- Prevents OS from moving thread between cores
- Keeps L1/L2 cache hot (no cold cache misses after migration)
- Reduces cache coherency traffic between cores
- Predictable performance (eliminates jitter from migrations)
- Enables NUMA-aware memory allocation

Linux: Setting CPU Affinity

On Linux, you control thread affinity using the `pthread_setaffinity_np` function. This is a non-portable POSIX extension (hence the `_np` suffix) that gives you precise control over which CPU cores a thread can run on.

```

#include <pthread.h>
#include <sched.h>

void pin_thread_to_core(int core_id) {
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(core_id, &cpuset);

    pthread_t current_thread = pthread_self();
    int result = pthread_setaffinity_np(
        current_thread,
        sizeof(cpu_set_t),
        &cpuset
    );

    if (result != 0) {
        throw std::runtime_error("Failed to pin thread");
    }

    // Verify pinning worked
    int actual_core = sched_getcpu();
    if (actual_core != core_id) {
        throw std::runtime_error("Thread pinned but running on wrong core");
    }
}

// Usage: core IDs come from configuration (isolated cores), never 0,
// which runs housekeeping and interrupt work
void worker_thread(int core_id) {
    pin_thread_to_core(core_id);

    // Now this thread runs only on core_id
    // ... low-latency work ...
}

```

Verification: The `sched_getcpu()` function returns the CPU core number the calling thread is currently executing on. This is useful for verifying that your thread is actually running on the expected core, especially when debugging performance issues or validating your pinning strategy.

NUMA Awareness

NUMA (Non-Uniform Memory Access) systems have multiple memory nodes, each with its own set of CPU cores. Memory access is faster when the CPU accesses memory on its local node versus a remote node (remote DRAM latency is typically ~1.5-2x local, and cross-socket cache-line transfers are slower still). For low-latency systems, you want to ensure threads access memory on their local NUMA node, and that the NIC is attached to the same node as the threads that use it.

```

#include <numa.h>

void pin_to_numa_node(int node_id) {
    if (numa_available() < 0) {
        throw std::runtime_error("NUMA not available");
    }

    // Run on this node's CPUs and allocate from its memory

```

```

    struct bitmask* mask = numa_allocate_nodemask();
    numa_bitmask_setbit(mask, node_id);
    numa_bind(mask);
    numa_free_nodemask(mask);
}

// Allocate memory on specific NUMA node
void* allocate_on_node(size_t size, int node) {
    return numa_alloc_onnode(size, node);
}

```

Setting Thread Priority

Real-time thread priorities prevent the OS from preempting your latency-critical threads for lower-priority work. `SCHED_FIFO` scheduling lets your thread run until it blocks or explicitly yields, with no time-slicing. This requires elevated privileges (root or `CAP_SYS_NICE` capability).

Be careful combining this with busy-polling. A thread that spins forever at a real-time priority starves every normal kernel thread on that core (per-CPU workqueues, RCU callbacks), which can cause soft-lockup warnings, RCU stalls or a hung system. Linux's RT throttling (`sched_rt_runtime_us`, 950ms per 1s by default) exists to stop this, and it periodically *pauses* your thread for 50ms, which is terrible for latency. The usual setup for spinning threads is `SCHED_OTHER` on an isolated core (`isolcpus/nohz_full/rcu_nocbs`) so nothing else wants to run there. Use `SCHED_FIFO` for threads that block, and avoid priority 99, which is shared with the kernel's own critical threads.

```

#include <pthread.h>

void set_realtime_priority() {
    struct sched_param param;
    param.sched_priority = 80; // High, but below kernel-critical threads (99)

    int result = pthread_setschedparam(
        pthread_self(),
        SCHED_FIFO, // Real-time FIFO scheduling
        &param
    );

    if (result != 0) {
        // Requires root/CAP_SYS_NICE
        throw std::runtime_error("Failed to set RT priority");
    }
}

```

Avoiding Context Switches

Context switches occur when the OS scheduler stops one thread and starts another, requiring saving and restoring CPU state. The switch itself takes about 1-2 microseconds; the larger, indirect cost is that the other thread evicts your data from the caches and TLB, so you run slowly for a while afterwards. Together this is 1-10 microseconds of latency - deadly for systems targeting sub-microsecond response times. The key to avoiding them is keeping threads running continuously without blocking on system calls or mutexes.

Busy-Wait (Spin) vs Sleep

The fundamental choice in waiting: busy-waiting burns CPU cycles checking a condition repeatedly but responds instantly when it becomes true. Sleeping yields the CPU to other threads but requires a context switch to wake up, adding microseconds of latency. For low-latency systems where dedicated cores are available, busy-waiting is almost always the right choice.

```
std::atomic<bool> data_ready{false};

// BAD: Sleep causes context switch (microseconds)
void wait_for_data_slow() {
    while (!data_ready.load(std::memory_order_acquire)) {
        std::this_thread::sleep_for(std::chrono::microseconds(1));
        // Actually sleeps ~50-60 s: Linux adds up to 50 s of timer
        // slack to normal threads' sleeps (prctl(PR_SET_TIMERSLACK)),
        // plus the context switch in and out
    }
}

// GOOD: Busy-wait (spin) - no context switch
void wait_for_data_fast() {
    while (!data_ready.load(std::memory_order_acquire)) {
        _mm_pause(); // Hint to CPU: we're spinning
        // One PAUSE is ~10 cycles on older Intel cores and ~140 cycles
        // (~40ns) on Skylake and later, so it adds that much to reaction time
    }
}

// Hybrid approach for non-critical paths: responsive when the wait is
// short, doesn't burn a core when it's long
void wait_hybrid() {
    // Spin briefly first
    for (int i = 0; i < 1000; ++i) {
        if (data_ready.load(std::memory_order_acquire)) {
            return;
        }
        _mm_pause();
    }

    // Then sleep if still not ready
    while (!data_ready.load(std::memory_order_acquire)) {
        std::this_thread::sleep_for(std::chrono::microseconds(10));
    }
}
```

CPU Pause Instruction

The PAUSE instruction (on x86; YIELD/ISB on ARM) is a hint to the CPU that we're in a spin-wait loop. It:

- **Avoids a pipeline flush on exit.** Without it, the CPU speculatively runs many loop iterations ahead; when the watched value finally changes, it detects a memory-ordering violation and flushes the pipeline, delaying exactly the moment you care about.
- **Frees resources for the hyperthread sibling** sharing the core.
- **Reduces power** while spinning.

Its latency varies a lot by microarchitecture: about 10 cycles before Skylake, about 140 cycles on Skylake and later Intel cores. Spin loops with a fixed iteration count (like the hybrid wait above) therefore spin for very different lengths of time on different machines; bound them by elapsed time (TSC) instead if it matters.

The `cpu_relax()` helper defined with the `SpinLock` above wraps `_mm_pause()`, and the spinlock uses it in its inner wait loop.

Interview Questions & Answers

1. “What’s the difference between a spin lock and a mutex? When would you use each?”

Answer:

Spin Lock: - Busy-waits in a loop checking if lock is available - Never voluntarily gives up the CPU (it can still be preempted) - Keeps thread active and cache hot - Very fast when contention is low and critical section is short - Implementation: test-and-test-and-set, as in the `SpinLock` class above

Mutex: - Uncontended: one atomic instruction in user space, about the same cost as a spinlock - Contended: asks the OS to put the thread to sleep (futex) and wake it later - The sleep/wake costs a context switch (1-10 microseconds) - Frees up CPU for other work; better for longer waits

When to use Spin Lock: - Critical section < 100 nanoseconds - Very low contention expected - Real-time system where you can’t afford context switch - You have dedicated CPU cores for your threads

When to use Mutex: - Critical section > 1 microsecond - High contention possible - General-purpose code - Don’t want to waste CPU cycles

Hybrid Approach: Spin briefly, then sleep. glibc’s `PTHREAD_MUTEX_ADAPTIVE_NP` mutex type does exactly this, so you rarely need to write it yourself.

Real-world example: In a trading system, the best lock on the hot path is no lock: give each order book to a single thread and feed it through SPSC queues. Where sharing is unavoidable and the critical section is a few nanoseconds on dedicated cores, a spinlock fits; for anything that does file or network I/O while holding the lock, use a mutex.

2. “Explain the ABA problem in lock-free programming. How do you solve it?”

Answer:

The ABA Problem:

The ABA problem occurs when a value changes from A to B and back to A, but a compare-and-swap (CAS) operation can’t detect that anything changed.

Example scenario:

```
// Lock-free stack
template<typename T>
class Stack {
    struct Node {
        T data;
        Node* next;
    };

    std::atomic<Node*> head;

    void pop() {
        Node* old_head = head.load();
        // PROBLEM: What if head changes here?
        head.compare_exchange_strong(old_head, old_head->next);
        delete old_head;
    }
};
```

```

    }
};

// Timeline:
// Thread 1: Reads head (node A)
// Thread 2: Pop A, Pop B, Push A (head is A again!)
// Thread 1: CAS succeeds (A == A), but A->next is stale!
//           Now points to freed memory!

```

The Problem Illustrated:

Initial: Stack has A -> B -> C

Thread 1: Read head (A)	Thread 2: Pop A (head = B) Pop B (head = C) Push A (head = A) A->next now points to C, not B!
------------------------------------	--

CAS(head, A, B)
Success! But B was freed!
Now head->next is freed memory

Solutions:

Solution 1: Tagged Pointers (Version Counter)

```

template<typename T>
class LockFreeStack {
    struct Node {
        T data;
        Node* next;
    };

    // Pack pointer and version counter
    struct TaggedPointer {
        Node* ptr;
        uintptr_t tag;
    };

    std::atomic<TaggedPointer> head{{nullptr, 0}};

public:
    void push(const T& data) {
        Node* new_node = new Node{data, nullptr};
        TaggedPointer old_head = head.load();

        do {
            new_node->next = old_head.ptr;
        } while (!head.compare_exchange_weak(
            old_head,
            {new_node, old_head.tag + 1} // Increment tag!
        ));
    }
}

```

```

bool pop(T& result) {
    TaggedPointer old_head = head.load();

    do {
        if (old_head.ptr == nullptr) return false;
    } while (!head.compare_exchange_weak(
        old_head,
        {old_head.ptr->next, old_head.tag + 1} // Increment tag!
    ));

    result = old_head.ptr->data;
    delete old_head.ptr; // Still need memory reclamation!
    return true;
}
};

```

The tag makes the CAS fail if anything happened in between, which fixes ABA. It does not fix use-after-free: `old_head.ptr->next` is still read from a node another thread may have deleted, so tagged pointers are combined with a reclamation scheme or with nodes that are never returned to the OS (a type-stable free list).

`std::atomic<TaggedPointer>` is 16 bytes. x86-64 can CAS 16 bytes with `cmpxchg16b`, but GCC routes 16-byte atomics through `libatomic` and reports them as not lock-free, even with `-mcx16`. The common alternative packs a 16-bit tag into the unused upper bits of a 64-bit pointer.

Solution 2: Hazard Pointers

```

// Proper hazard pointer implementation
class HazardPointer {
    static constexpr int MAX_THREADS = 128;

    struct HazardRecord {
        std::atomic<std::thread::id> owner{}; // Which thread owns this slot
        std::atomic<Node*> hazard{nullptr}; // Protected pointer
    };

    std::array<HazardRecord, MAX_THREADS> hazards_;

public:
    // Each thread acquires a hazard record
    HazardRecord* acquire_record() {
        std::thread::id tid = std::this_thread::get_id();

        // Try to find existing record for this thread
        for (auto& rec : hazards_) {
            std::thread::id expected{};
            if (rec.owner.load(std::memory_order_acquire) == tid ||
                rec.owner.compare_exchange_strong(expected, tid,
                                                    std::memory_order_release)) {
                return &rec;
            }
        }
        throw std::runtime_error("No hazard records available");
    }

    // Thread announces it's using this pointer.

```

```

// Must be seq_cst: the caller re-reads head_ right after this, and the
// hazard store must be visible to other threads before that load.
// Release/acquire would allow StoreLoad reordering (even on x86), so a
// reclaiming thread could miss the hazard and free the node.
void protect(HazardRecord* rec, Node* ptr) {
    rec->hazard.store(ptr, std::memory_order_seq_cst);
}

// Thread done with this pointer
void clear(HazardRecord* rec) {
    rec->hazard.store(nullptr, std::memory_order_release);
}

// Check if any thread is using this pointer
bool is_protected(Node* ptr) {
    for (auto& rec : hazards_) {
        if (rec.hazard.load(std::memory_order_seq_cst) == ptr) {
            return true;
        }
    }
    return false;
}

// Defer deletion; free retired nodes in batches that no hazard protects.
// Scanning only every 2 * MAX_THREADS retirements keeps the cost amortized O(1).
void retire(Node* ptr) {
    thread_local std::vector<Node*> retired;
    retired.push_back(ptr);
    if (retired.size() >= 2 * MAX_THREADS) {
        std::erase_if(retired, [this](Node* p) { // C++20
            if (is_protected(p)) return false; // keep for later
            delete p;
            return true;
        });
    }
}

};

// Usage in lock-free stack
bool pop(T& result, HazardPointer& hp) {
    HazardRecord* rec = hp.acquire_record();

    Node* old_head;
    do {
        old_head = head_.load(std::memory_order_acquire);
        if (!old_head) {
            hp.clear(rec);
            return false;
        }

        // Protect the node we're about to use
        hp.protect(rec, old_head);

        // Verify head hasn't changed since we read it: if it hasn't, the

```

```

    // node wasn't retired before our hazard became visible
    if (head_.load(std::memory_order_seq_cst) != old_head) {
        continue; // Retry
    }

    // Now safe to dereference old_head
    if (head_.compare_exchange_weak(old_head, old_head->next,
                                    std::memory_order_release,
                                    std::memory_order_acquire)) {
        result = old_head->data;
        hp.clear(rec); // Done with this pointer
        hp.retire(old_head); // Safe to delete later
        return true;
    }
} while (true);
}

```

How this prevents the race: Each thread has its own slot in the hazard array (identified by `thread::id`); in real code, cache the record in a `thread_local` rather than scanning for it on every call. To use a node, a thread publishes it in its slot, then re-checks that the node is still reachable (`head_ unchanged`). A reclaiming thread removes a node from the structure *before* retiring it, and scans all slots before freeing. Because both the hazard store and the re-check are `seq_cst`, either the reclaimer sees the hazard (and keeps the node) or the reader sees that `head_ changed` (and retries). The node can't be freed while it's being dereferenced.

Solution 3: Epoch-Based Reclamation (RCU-like)

```

// Sketch: each thread announces the epoch it entered in; a node retired in
// epoch e can be freed once every active thread has entered a later epoch.
class EpochManager {
    static constexpr uint64_t IDLE = UINT64_MAX;
    static constexpr int MAX_THREADS = 128;

    std::atomic<uint64_t> global_epoch{0};
    // One slot per registered thread. It must be a shared array, not a
    // thread_local, because the reclaimer has to read every thread's epoch.
    std::array<std::atomic<uint64_t>, MAX_THREADS> thread_epochs_;

public:
    void enter(int tid) {
        // seq_cst: the announcement must be visible before we read
        // any pointers from the data structure
        thread_epochs_[tid].store(global_epoch.load(), std::memory_order_seq_cst);
    }

    void exit(int tid) {
        thread_epochs_[tid].store(IDLE, std::memory_order_release);
    }

    void retire(Node* ptr) {
        // Save with current epoch
        retired_nodes.push_back({ptr, global_epoch.load()});
    }

    void try_reclaim() {
        // Find minimum epoch across all active threads (scan thread_epochs_)
    }
}

```

```

uint64_t min_epoch = find_min_active_epoch();

// Delete nodes from epochs before min_epoch
for (auto it = retired_nodes.begin(); it != retired_nodes.end(); ) {
    if (it->epoch < min_epoch) {
        delete it->ptr;
        it = retired_nodes.erase(it);
    } else {
        ++it;
    }
}

// Advance epoch periodically
if (should_advance()) {
    global_epoch.fetch_add(1);
}
};

```

Key Interview Points: - ABA is specific to CAS-based algorithms - ABA and use-after-free are separate problems; reclamation schemes solve both, tags only ABA - Version counters are simplest; a 64-bit counter never wraps in practice, a 16-bit one packed into pointer bits can - Hazard pointers are precise and bound unreclaimed memory, but cost a seq_cst store per protected access - Epoch-based is fastest for readers but delays reclamation, and one stalled thread blocks all reclamation (unbounded memory growth) - Modern: Use proven libraries like libcds or folly::hazptr; C++26 standardizes std::hazard_pointer and RCU (std::rcu_obj_base)

3. “When would you use memory_order_relaxed vs memory_order_acquire?”

Answer:

The memory ordering determines what synchronization guarantees you get with atomic operations. Choosing the right one is crucial for both correctness and performance.

memory_order_relaxed:

Use when you only need atomicity, not synchronization between threads.

When to use: - Simple counters where you don't care about ordering - Statistics collection - When no other memory operations depend on the atomic operation

```

// Example: Simple counter
std::atomic<int64_t> message_count{0};

void on_message_received() {
    // Just increment, don't care about order
    message_count.fetch_add(1, std::memory_order_relaxed);
}

// Reading the count
int64_t get_count() {
    return message_count.load(std::memory_order_relaxed);
}

```

Performance: A relaxed load is a plain mov; a relaxed fetch_add is a lock xadd on x86, the same instruction (and cost) as a seq_cst one. On x86 the benefit of relaxed is mainly that it lets the compiler

reorder around it; on ARM it also avoids barrier instructions.

memory_order_acquire (and memory_order_release):

Use when you need synchronization - when one thread needs to see the side effects of another thread's operations.

When to use: - Publisher-subscriber patterns - Lazy initialization - When non-atomic data must be synchronized - Implementing lock-free data structures

```
// Example: Publishing data once (calling publish() twice while a consumer
// reads would race on `data`)
class DataPublisher {
    std::string data; // Non-atomic!
    std::atomic<bool> ready{false};

public:
    // Producer thread
    void publish(const std::string& new_data) {
        data = new_data; // Write data first
        // Release: Ensures all writes above are visible
        ready.store(true, std::memory_order_release);
    }

    // Consumer thread
    std::optional<std::string> try_consume() {
        // Acquire: Ensures we see the data write
        if (ready.load(std::memory_order_acquire)) {
            return data; // Safe to read
        }
        return std::nullopt;
    }
};
```

The guarantee: All memory writes before a **release** operation are visible to a thread that performs an **acquire** operation that sees that value.

Real-world example - Lock-free queue: The SPSC queue earlier in this chapter uses all three: **relaxed** to read its own index, and two separate release/acquire pairs. The producer's release of **tail_** publishes the item to the consumer. The consumer's release of **head_** guarantees it has finished reading a slot before the producer reuses it. The second pair is the one people forget.

Quick decision tree: - Just counting/stats with no dependencies? → **relaxed** - Protecting non-atomic data? → **acquire/release** - Need a single total order across several atomics (e.g. Dekker-style "I set my flag, then check yours")? → **seq_cst** - Lock-free algorithm from a paper? → Follow the paper's specifications exactly

Performance difference on x86: - Loads: **relaxed**, **acquire** and **seq_cst** are all a plain **mov** - Stores: **relaxed** and **release** are a plain **mov**; **seq_cst** is **xchg** (~20-30 cycles) - Read-modify-writes: all orderings are the same lock-prefixed instruction - The weaker orderings still matter on x86 because they allow more *compiler* reordering

On ARM or other weakly-ordered architectures, **acquire** and **release** need barrier or special load/store instructions, and the differences are more pronounced.

4. “How would you implement a lock-free queue?”

Answer:

I’d start by clarifying requirements: Single-producer single-consumer (SPSC), multi-producer single-consumer (MPSC), or multi-producer multi-consumer (MPMC)? Each has different complexity.

For SPSC (most common in low-latency trading):

Write the SPSC queue from earlier in this chapter, and explain these design decisions:

1. **Power-of-2 capacity:**
 - Allows using bitwise AND instead of modulo (no division)
 - $(\text{index} + 1) \& \text{MASK}$ instead of $(\text{index} + 1) \% \text{Capacity}$
2. **Cache line alignment:**
 - Separates head and tail to avoid false sharing
 - Producer only writes tail, consumer only writes head
3. **Memory ordering:**
 - relaxed for own indices (no other thread writes them)
 - acquire when reading other thread’s index (need synchronization)
 - release when publishing updates: the producer’s publishes the item, the consumer’s publishes that the slot is free again
4. **One-slot wasted:**
 - When $\text{next_tail} == \text{head}$, queue is full
 - This distinguishes full from empty (both would have $\text{tail} == \text{head}$)
5. **Optimization:** cache the other side’s index locally and re-read it only when the queue looks full/empty

For MPSC (Multi-Producer Single-Consumer):

Dmitry Vyukov’s intrusive MPSC queue: producers link nodes in with a single atomic `exchange` (no CAS loop), and the consumer walks the list from a dummy “stub” node:

```
template<typename T>
class MPSCQueue {
    struct Node {
        std::atomic<Node*> next{nullptr};
        T data;
    };

    alignas(64) std::atomic<Node*> head_; // consumer side
    alignas(64) std::atomic<Node*> tail_; // producer side

public:
    MPSCQueue() {
        Node* stub = new Node{}; // dummy node: head_ and tail_ are never null
        head_.store(stub, std::memory_order_relaxed);
        tail_.store(stub, std::memory_order_relaxed);
    }

    ~MPSCQueue() { // call only once producers have stopped
        T item;
        while (try_pop(item)) {}
        delete head_.load(std::memory_order_relaxed); // the current stub
    }

    // Multiple producers can call this
    void push(const T& item) {
        Node* node = new Node{nullptr, item};
```

```

    // Atomically swap in our node as the new tail
    Node* prev_tail = tail_.exchange(node, std::memory_order_acq_rel);

    // Link previous tail to our node
    prev_tail->next.store(node, std::memory_order_release);
}

// Only one consumer
bool try_pop(T& item) {
    Node* head = head_.load(std::memory_order_relaxed);
    Node* next = head->next.load(std::memory_order_acquire);

    if (next == nullptr) {
        return false; // Empty
    }

    item = next->data; // `next` becomes the new stub
    head_.store(next, std::memory_order_release);
    delete head; // In production, use a pool
    return true;
}
};

```

Between a producer's `exchange` and its `next.store`, the list is briefly broken: the consumer sees “empty” even though later items exist. If that producer is preempted there, the consumer stalls until it resumes, so this queue is not strictly lock-free. It's still a common, very fast choice.

For SPMC (Single-Producer Multi-Consumer):

First ask whether each item goes to *one* consumer (work distribution) or to *all* of them (broadcast), since the designs differ. See SPMC Queues: a single-producer variant of Vyukov's queue, or per-consumer SPSC queues, for work distribution; and for broadcast, a Disruptor-style ring (producer waits for the slowest consumer) or a seqlock ring (producer never waits, slow consumers detect overruns).

Challenges to mention: - ABA problem in MPMC queues - Memory reclamation (hazard pointers or epoch-based reclamation) - Ensuring progress guarantees (lock-free vs wait-free)

Testing approach: - Stress test with multiple threads - Use thread sanitizer to detect data races - Verify no ABA issues - Measure latency under contention

5. “How would you implement thread pinning for a low-latency trading system?”

Answer:

Thread pinning (CPU affinity) is critical for low-latency systems to maintain cache locality and avoid scheduler overhead.

Basic Implementation: `pin_thread_to_core()` from Setting CPU Affinity above: `pthread_setaffinity_np` with a one-CPU mask, then verify with `sched_getcpu()`.

To keep network interrupts off pinned cores, see **IRQ affinity in Chapter 4**.

Key Interview Points: - Pin critical threads to isolated cores (`isolcpus=`, plus `nohz_full=` and `rcu_nocbs=` to remove the timer tick and RCU callbacks from them) - Avoid hyperthreads for latency-critical threads: leave the sibling logical CPU idle, or disable SMT - Keep related threads, their memory and the NIC on the same NUMA node - On isolated cores, a spinning thread doesn't need a real-time

priority (and a spinning SCHED_FIFO thread can starve kernel work; see Setting Thread Priority) - Verify pinning with `taskset -p <pid>` or `ps -eLo pid,tid,psr,comm`

Common Mistakes: - Pinning two busy threads to hyperthread siblings (they share the core's execution units and L1/L2 caches, so each slows the other) - Ignoring NUMA topology (remote memory access penalties) - Forgetting to disable CPU frequency scaling (`cpupower`) - Not isolating cores from OS scheduler

6. “How do you debug race conditions and data races in concurrent code?”

Answer:

Tools and Techniques:

1. Thread Sanitizer (TSan)

```
# Compile with TSan
g++ -fsanitize=thread -g -O2 program.cpp

# Run
./a.out

# Output shows:
# WARNING: ThreadSanitizer: data race
#   Write of size 4 at 0x7fff12345678 by thread T2:
#     #0 increment() program.cpp:42
#   Previous write of size 4 at 0x7fff12345678 by thread T1:
#     #0 increment() program.cpp:42
```

2. Helgrind (Valgrind tool)

```
valgrind --tool=helgrind ./program

# Detects:
# - Lock order violations
# - Data races
# - Misuse of POSIX pthreads API
```

3. Intel Inspector

```
inspxe-cl -collect ti3 -result-dir results ./program
# Detects data races and deadlocks
```

4. Address Sanitizer for use-after-free

```
# Compile with ASan
g++ -fsanitize=address -g program.cpp

# Catches:
# - Use after free
# - Double free
# - Memory leaks
```

Manual Debugging Techniques:

Logging with Ordering:

```
#include <atomic>
#include <cinttypes>
#include <cstdio>
```

```

#include <unistd.h>

std::atomic<uint64_t> global_counter{0};

void debug_log(const char* msg) {
    uint64_t order = global_counter.fetch_add(1);
    printf("[%u PRIu64 "] Thread %d: %s\n",
           order,
           static_cast<int>(gettid()), // std::thread::id is not an integer
           msg);
}

```

// Reconstruct exact ordering of events across threads

Beware that logging changes timing, and `printf` takes a lock, which adds synchronization between threads. Adding logging can make a race disappear. For low-overhead tracing, write fixed-size records into a per-thread ring buffer and dump them afterwards.

Stress Testing:

```

void stress_test() {
    const int NUM_THREADS = 128;
    const int ITERATIONS = 1000000;

    std::vector<std::thread> threads;

    for (int i = 0; i < NUM_THREADS; ++i) {
        threads.emplace_back([&]() {
            for (int j = 0; j < ITERATIONS; ++j) {
                // Exercise concurrent code
                test_operation();
            }
        });
    }

    for (auto& t : threads) {
        t.join();
    }

    // Verify invariants
    assert(verify_consistency());
}

```

Happens-Before Analysis:

```

// Document synchronization points
class OrderBook {
    std::mutex mutex_;
    std::vector<Order> orders_;

public:
    void add_order(const Order& order) {
        std::lock_guard lock(mutex_); // Synchronization point
        orders_.push_back(order);
        // Happens-before relationship established
    }
}

```

```

    Order get_order(size_t idx) {
        std::lock_guard lock(mutex_); // Synchronization point
        return orders_[idx];
        // Will see all modifications before previous unlock
    }
};

```

Reproduce Race Conditions:

```

// Add strategic delays to expose races
void suspicious_function() {
    read_data();

    std::this_thread::sleep_for(std::chrono::microseconds(100)); // Widen race window

    write_data();
}

// Or use random delays
void add_random_delay() {
    static thread_local std::mt19937 gen(std::random_device{}());
    std::uniform_int_distribution<> dis(0, 1000);
    std::this_thread::sleep_for(std::chrono::nanoseconds(dis(gen)));
}

```

Key Interview Points: - Always use ThreadSanitizer in testing - Understand happens-before relationships - Document all synchronization points - Stress test with more threads than cores - Use assertions liberally to check invariants - Race conditions are often timing-dependent - test repeatedly

7. “Explain how you would implement a wait-free queue and why it matters.”

Answer:

Wait-free vs Lock-free: - **Lock-free:** The system as a whole always makes progress: whenever threads are running, some operation completes in a finite number of steps (an individual thread may starve) - **Wait-free:** Every thread completes its operation in a bounded number of its own steps, regardless of what other threads do (stronger guarantee)

Wait-free matters in real-time systems where you need guaranteed latency bounds for every operation, not just on average.

Wait-Free SPSC Queue:

The SPSC queue earlier in this chapter is already wait-free: `try_push` and `try_pop` each do a fixed number of loads and stores and return immediately when the queue is full or empty.

Why It Is Wait-Free: - No CAS loops (no retry on failure) - No spinning/waiting inside the operation - Fixed number of atomic operations - Bounded execution time: $O(1)$ worst case

(A caller that spins on `try_pop` until it succeeds is waiting, of course; wait-freedom is a property of the operation, not of how you use it.)

Wait-Free MPMC Queue (More Complex):

For MPMC, we need something like Kogan-Petrunkin’s wait-free queue or use helping mechanisms:

```

// Pseudocode sketch of the "announce and help" idea (Operation,
// announce_enqueue, help_complete etc. are not shown; the real algorithm

```

```

// is considerably more complex)
template<typename T>
class WaitFreeMPMCQueue {
    struct Node {
        std::atomic<T*> data{nullptr};
        std::atomic<Node*> next{nullptr};
    };

    struct HelpRecord {
        std::atomic<Operation*> pending_op{nullptr};
    };

    std::array<HelpRecord, MAX_THREADS> help_records_;

    // Each operation helps others if needed
    void enqueue(const T& item, int thread_id) {
        // 1. Announce operation
        Operation* op = announce_enqueue(item, thread_id);

        // 2. Help other threads
        help_all_pending();

        // 3. Execute own operation
        execute_enqueue(op);

        // 4. Help again if needed
        help_all_pending();

        // Bounded number of steps!
    }

    void help_all_pending() {
        // Check all threads' pending operations
        for (auto& record : help_records_) {
            Operation* op = record.pending_op.load();
            if (op != nullptr) {
                help_complete(op); // Help them finish
            }
        }
    }
};

```

Practical Considerations:

When to use wait-free: - Hard real-time systems - Safety-critical applications - When worst-case latency matters more than average

Trade-offs: - More complex to implement - Often slower average case than lock-free - May require more memory (helping records) - SPSC is easy, MPMC is very complex

Interview Answer Summary:

“I’d implement a wait-free queue when I need guaranteed latency bounds for every operation. For SPSC, it’s straightforward - no CAS loops needed, just atomic loads and stores with proper memory ordering. For MPMC, it’s much more complex and typically requires helping mechanisms where threads help each other complete operations. In practice, low-latency trading systems are built from SPSC queues, which are wait-

free for free; for multi-producer cases a well-implemented lock-free or Vyukov-style queue is usually sufficient, and true wait-free MPMC is reserved for hard real-time systems where a worst-case bound must be proven.”

Practical Examples

Complete Low-Latency Pipeline

```
class TradingPipeline {
    // Market data thread -> strategy thread -> order thread,
    // connected by SPSC queues (one producer, one consumer each)

    SPSCQueue<MarketData, 4096> market_data_queue_;
    SPSCQueue<Signal, 1024> signal_queue_;

    std::atomic<bool> running_{true};
    std::atomic<uint64_t> dropped_md_{0};

    void market_data_thread() {
        pin_thread_to_core(2); // isolated cores, from configuration

        while (running_.load(std::memory_order_relaxed)) {
            MarketData data;
            if (!poll_market_data(data)) continue; // busy-poll the NIC
            if (!market_data_queue_.try_push(data)) {
                // Full: the strategy can't keep up. Never ignore this -
                // count it, and resynchronize the book from a snapshot
                dropped_md_.fetch_add(1, std::memory_order_relaxed);
            }
        }
    }

    void strategy_thread() {
        pin_thread_to_core(3);

        MarketData data;
        while (running_.load(std::memory_order_relaxed)) {
            if (market_data_queue_.try_pop(data)) {
                Signal signal = compute_signal(data);
                if (signal.should_trade && !signal_queue_.try_push(signal)) {
                    handle_backpressure(signal); // e.g. drop and alert
                }
            }
        }
    }

    void order_thread() {
        pin_thread_to_core(4);

        Signal signal;
        while (running_.load(std::memory_order_relaxed)) {
            if (signal_queue_.try_pop(signal)) {
                send_to_exchange(create_order(signal));
            }
        }
    }
}
```

```
    }  
};
```

Every hand-off between cores costs a cache-line transfer (~50-100ns), so many systems run market data, the book and the strategy on a single thread and only hand off to a separate thread for things like logging.

Contents

Chapter 3: Modern C++ Features for Low-Latency Systems	1
Table of Contents	1
Move Semantics & Perfect Forwarding	2
Move Semantics Basics	2
Perfect Forwarding	3
Avoiding Unnecessary Copies	3
Return Value Optimization (RVO)	3
Template Metaprogramming	4
Compile-Time Computations	4
Type Traits for Zero-Cost Abstractions	5
SFINAE and Enable If	5
Variadic Templates	6
constexpr for Compile-Time Computation	6
constexpr Functions	6
constexpr Constructors	7
constexpr if (C++17)	7
constexpr Lookup Tables	7
std::variant and std::optional	8
std::optional for Error Handling	8
std::variant for Type-Safe Unions	9
Zero-Overhead Variant Processing	10
Smart Pointers in Low-Latency Code	10
When to Use Smart Pointers	10
std::unique_ptr - Zero Overhead	11
Avoiding shared_ptr in Hot Path	11
Ranges and Views (C++20)	12
Range Views for Zero-Copy Operations	12
Custom Ranges	13
Concepts (C++20)	13
Type Constraints	13
Concepts for Better Error Messages	14
Interview Questions & Answers	15
1. “Explain move semantics and when they improve performance.”	15
2. “What is perfect forwarding and why is it useful?”	16
3. “Explain constexpr and its benefits for low-latency code.”	17
4. “When would you use std::variant over inheritance?”	20
5. “How do you use templates to achieve zero-cost abstractions?”	22
Summary	28

Chapter 3: Modern C++ Features for Low-Latency Systems

This chapter explores modern C++ features (C++11 through C++20) that are essential for building high-performance, low-latency trading systems. We focus on features that provide zero-cost abstractions, eliminate runtime overhead, and enable compile-time optimization—all critical for achieving microsecond-level latency requirements.

Table of Contents

1. Move Semantics & Perfect Forwarding
2. Template Metaprogramming
3. constexpr for Compile-Time Computation
4. std::variant and std::optional

5. Smart Pointers in Low-Latency Code
 6. Ranges and Views (C++20)
 7. Concepts (C++20)
 8. Interview Questions & Answers
-

Move Semantics & Perfect Forwarding

Move semantics revolutionized C++ by allowing efficient transfer of resources instead of expensive copying. In low-latency systems where microseconds matter, eliminating unnecessary memory allocations and copies can dramatically improve performance. This section covers how to leverage rvalue references, move constructors, and perfect forwarding to achieve zero-copy semantics.

Move Semantics Basics

Move semantics allow you to transfer ownership of resources from temporary or expiring objects, avoiding expensive deep copies. This is particularly important for managing large buffers, order books, or market data structures where copying would incur significant latency.

```
class Order {
    std::vector<char> data_;

public:
    Order() = default;

    // Copy constructor (expensive)
    Order(const Order& other) : data_(other.data_) {
        // Deep copy - allocates memory
    }

    // Move constructor (cheap)
    Order(Order&& other) noexcept : data_(std::move(other.data_)) {
        // Just steal the pointer - no allocation!
    }

    // Copy assignment
    Order& operator=(const Order& other) {
        data_ = other.data_; // Deep copy (vector handles self-assignment)
        return *this;
    }

    // Move assignment
    Order& operator=(Order&& other) noexcept {
        data_ = std::move(other.data_); // Steal resources
        return *this;
    }
};

// These four are exactly what the compiler would generate. In real code,
// write none of them ("rule of zero") and the defaults do the right thing,
// including noexcept on the moves.

// Usage
Order create_order() {
```

```

    Order order;
    // ... populate ...
    return order; // NRVO usually; at worst an implicit move, never a copy
}

Order order = create_order(); // C++17: no copy or move here at all -
                             // the prvalue initializes `order` directly

```

Perfect Forwarding

Perfect forwarding preserves the value category (lvalue vs rvalue) of arguments when passing them through template functions. This enables zero-overhead wrapper functions and factory methods that forward arguments without introducing copies, essential for maintaining low latency in abstraction layers.

```

// Forward arguments without copying
template<typename... Args>
auto make_order(Args&&... args) {
    return std::make_unique<Order>(std::forward<Args>(args)...);
    // Preserves lvalue/rvalue-ness
}

// Usage
auto order1 = make_order(std::string("AAPL"), 150.0); // Rvalues: moved into Order
std::string sym = "AAPL";
auto order2 = make_order(sym, 150.0);                // Lvalue: copied, sym untouched

```

Avoiding Unnecessary Copies

In hot paths of trading systems, even a single unnecessary copy can add hundreds of nanoseconds. Understanding when to pass by value, reference, or rvalue reference is crucial for minimizing latency. This subsection demonstrates common patterns and anti-patterns.

```

// BAD: Copies the whole vector (and every Order in it) on each call
void process_orders_bad(std::vector<Order> orders) {
    for (Order order : orders) { // ...and copies each Order again
        send_to_exchange(order);
    }
}

// GOOD: Pass by const reference (or std::span<const Order>)
void process_orders(const std::vector<Order>& orders) {
    for (const Order& order : orders) {
        send_to_exchange(order); // void send_to_exchange(const Order&)
    }
}

// When the callee needs to keep the object, move ownership in
void enqueue_order(Order order); // by value: caller decides copy or move

Order o = build_order();
enqueue_order(std::move(o)); // moved, no copy (o is now empty)

```

Return Value Optimization (RVO)

Compilers construct return values directly in the caller's storage, skipping the copy or move. There are two cases: - **RVO** (returning a temporary, `return Order{...};`): guaranteed since C++17, and works even

for types that can't be copied or moved. - **NRVO** (returning a named local, `return order;`): allowed but not guaranteed. Compilers do it when every return statement returns the same local; otherwise the local is implicitly moved.

```
// RVO: guaranteed, no copy or move
Order create_order() {
    return Order{"AAPL", 150.0};
}

// NRVO: a single named object returned on every path
Order create_complex_order(bool condition) {
    Order order; // Single named object
    if (condition) {
        order.set_type(BUY);
    } else {
        order.set_type(SELL);
    }
    return order; // NRVO applies
}

// NRVO blocked: different objects returned on different paths
Order pick(bool c) {
    Order a, b;
    return c ? a : b; // COPY: the ?: expression is not a plain local name
    // `if (c) return a; return b;` would at least move
}
```

Template Metaprogramming

Template metaprogramming allows computation and type manipulation at compile time, moving work from runtime to compilation. This technique is fundamental for achieving zero-cost abstractions where the compiler generates optimal code for each specific use case without runtime overhead.

Compile-Time Computations

Computing values at compile time using template recursion eliminates runtime calculations entirely. This is useful for buffer sizes, lookup table generation, and configuration validation—all determined before the program even runs.

```
// Compute at compile time
template<int N>
struct Factorial {
    static constexpr int value = N * Factorial<N-1>::value;
};

template<>
struct Factorial<0> {
    static constexpr int value = 1;
};

// Usage
constexpr int result = Factorial<5>::value; // 120, computed at compile time
std::array<int, Factorial<5>::value> buffer; // Size known at compile time
```

Type Traits for Zero-Cost Abstractions

Type traits enable compile-time introspection of types, allowing you to write generic code that automatically selects the optimal implementation path. In low-latency code, this means using fast `memcpy` for trivial types while maintaining correctness for complex types.

```
// Copy `count` objects into uninitialized storage at `dest`
template<typename T>
void fast_copy(T* dest, const T* src, size_t count) {
    if constexpr (std::is_trivially_copyable_v<T>) {
        // Fast path: use memcpy
        std::memcpy(dest, src, count * sizeof(T));
    } else {
        // Slow path: use copy constructor
        for (size_t i = 0; i < count; ++i) {
            new (&dest[i]) T(src[i]);
        }
    }
}

// (std::copy and std::uninitialized_copy already do this dispatch
// internally, so in real code just call them)

// Custom type trait
template<typename T>
struct is_fixed_size {
    static constexpr bool value = std::is_trivially_copyable_v<T> &&
        !std::is_pointer_v<T>;
};

template<typename T>
inline constexpr bool is_fixed_size_v = is_fixed_size<T>::value;
```

SFINAE and Enable If

SFINAE (Substitution Failure Is Not An Error) allows compile-time function overload selection based on type properties. This enables writing different implementations for different type categories without runtime dispatching, essential for performance-critical template code.

```
// Only enable for integral types
template<typename T>
std::enable_if_t<std::is_integral_v<T>, T>
safe_add(T a, T b) {
    T result;
    // Checks both directions (and unsigned wrap) in one instruction + jo/jc
    if (__builtin_add_overflow(a, b, &result)) {
        throw std::overflow_error("Addition overflow");
    }
    return result;
}

// Only enable for floating point
template<typename T>
std::enable_if_t<std::is_floating_point_v<T>, T>
safe_add(T a, T b) {
    // Different logic for floats
    return a + b;
}
```

```
}
```

Variadic Templates

Variadic templates handle an arbitrary number of type-safe arguments, enabling flexible interfaces without runtime overhead. Combined with fold expressions (C++17), they provide elegant solutions for logging, tuple operations, and generic factory functions.

```
// Process any number of arguments
template<typename... Args>
void log(Args&&... args) {
    (std::cout << ... << args) << '\n'; // Fold expression (C++17)
}

// Usage
log("Order: ", 123, " Price: ", 150.50);

// Recursive variadic processing
template<typename First, typename... Rest>
void process_args(First&& first, Rest&&... rest) {
    process_single(std::forward<First>(first));

    if constexpr (sizeof...(rest) > 0) {
        process_args(std::forward<Rest>(rest)...);
    }
}
```

constexpr for Compile-Time Computation

The `constexpr` specifier moves computation from runtime to compile time, resulting in zero runtime cost. For low-latency systems, this means lookup tables, configuration validation, and mathematical constants are all resolved during compilation, leaving only the minimal necessary work for runtime.

constexpr Functions

Functions marked `constexpr` can execute at compile time when given constant expressions, or at runtime when needed. This dual capability enables writing code once that optimizes automatically based on context—compile-time when possible, runtime when necessary.

```
// Computed at compile time if possible
constexpr int square(int x) {
    return x * x;
}

constexpr int cube(int x) {
    return x * x * x;
}

// Used in compile-time contexts
std::array<int, square(10)> buffer; // Size = 100, known at compile time
static_assert(cube(3) == 27);      // Compile-time assertion

// Can also be called at runtime
```

```
int runtime_val = get_input();
int result = square(runtime_val); // Computed at runtime
```

constexpr Constructors

Classes with `constexpr` constructors can be instantiated at compile time, enabling compile-time object creation and manipulation. This is particularly useful for configuration objects and mathematical types used in trading algorithms.

```
struct Point {
    int x, y;

    constexpr Point(int x_, int y_) : x(x_), y(y_) {}

    constexpr int distance_squared() const {
        return x * x + y * y;
    }
};

// Compile-time computation
constexpr Point p(3, 4);
constexpr int dist = p.distance_squared(); // 25, compile-time
static_assert(dist == 25);
```

constexpr if (C++17)

`constexpr if` enables compile-time branch selection, where only the taken branch is compiled. This eliminates runtime branching overhead and dead code, allowing generic code to specialize perfectly for each type instantiation.

```
template<typename T>
auto get_value(const T& container) {
    if constexpr (std::is_array_v<T>) {
        return container[0];
    } else if constexpr (requires { container.front(); }) { // requires-expression: C++20
        return container.front();
    } else {
        return *container.begin();
    }
    // Only one branch is instantiated for each T!
}
```

Discarded branches are only skipped inside templates, where they depend on `T`. In ordinary code, both branches of an `if constexpr` must still compile.

constexpr Lookup Tables

Pre-computing lookup tables at compile time replaces expensive computation with a memory access. This pays off for CRC calculations, hash functions, or functions that take many instructions per input.

A lookup is not free: an L1 hit costs ~4-5 cycles and a cache miss ~100ns, and the table takes cache space away from other data. For anything a few arithmetic instructions can compute (a multiply is 3 cycles), computing is faster than looking up. Measure before replacing arithmetic with a table.

```
#include <array>
#include <cstdint>
```

```

// Generate lookup table at compile time: bit reversal of a byte
// (10 instructions to compute, one load to look up)
constexpr auto generate_bit_reverse() {
    std::array<uint8_t, 256> table{};
    for (int i = 0; i < 256; ++i) {
        uint8_t r = 0;
        for (int b = 0; b < 8; ++b) {
            if (i & (1 << b)) r |= 1 << (7 - b);
        }
        table[i] = r;
    }
    return table;
}

// Lookup table is placed in read-only data
constexpr auto BIT_REVERSE = generate_bit_reverse();

// Fast runtime lookup; the uint8_t parameter makes out-of-range indices impossible
inline uint8_t reverse_bits(uint8_t x) {
    return BIT_REVERSE[x];
}

```

std::variant and std::optional

Modern C++ provides `std::variant` and `std::optional` as type-safe alternatives to unions and null pointers. These constructs offer both safety and performance—particularly valuable in low-latency systems where both correctness and speed are paramount.

std::optional for Error Handling

`std::optional` represents “maybe a value” without exceptions or sentinel values, and without heap allocation. Its cost is a bool flag (plus padding) next to the value, and returning `optional<T>` returns a *copy* of `T`.

```

#include <optional>

// Better than returning nullptr or throwing exceptions
std::optional<Order> find_order(int order_id) {
    auto it = orders.find(order_id);
    if (it != orders.end()) {
        return it->second;
    }
    return std::nullopt; // Not found
}

// Usage
if (auto order = find_order(123)) {
    process(*order); // Dereference to get value
} else {
    // Handle not found
}

// Or with value_or

```

```

Order order = find_order(123).value_or(default_order);

// Note: find_order copies the Order out of the map. If the caller only
// needs to look at it, return a pointer instead (optional<T&&> is only
// available from C++26):
const Order* find_order_ptr(int order_id) {
    auto it = orders.find(order_id);
    return it != orders.end() ? &it->second : nullptr;
}

```

std::variant for Type-Safe Unions

std::variant is a type-safe union that knows which alternative it currently holds. Unlike a class hierarchy with virtual functions, variants use value semantics with no heap allocation or virtual function calls, making them ideal for message types, order variants, and state machines in low-latency code.

```

#include <variant>

// Type-safe union
using OrderData = std::variant<
    MarketOrder,
    LimitOrder,
    StopOrder
>;

class Order {
    OrderData data_;

public:
    // Constrained: without the requires-clause, this forwarding constructor
    // is a better match than the copy constructor for a non-const Order
    // lvalue, so `Order b = a;` would try to build a variant from an Order
    template<typename T>
        requires std::is_constructible_v<OrderData, T&&>
    Order(T&& data) : data_(std::forward<T>(data)) {}

    // Visit pattern for processing
    void execute() {
        std::visit([](auto&& order) {
            using T = std::decay_t<decltype(order)>;
            if constexpr (std::is_same_v<T, MarketOrder>) {
                execute_market(order);
            } else if constexpr (std::is_same_v<T, LimitOrder>) {
                execute_limit(order);
            } else if constexpr (std::is_same_v<T, StopOrder>) {
                execute_stop(order);
            }
        }, data_);
    }

    // Check current type
    bool is_market_order() const {
        return std::holds_alternative<MarketOrder>(data_);
    }
}

```

```

    // Get value if type matches
    MarketOrder* get_if_market() {
        return std::get_if<MarketOrder>(&data_);
    }
};

```

Zero-Overhead Variant Processing

The visitor pattern combined with lambdas enables processing variants without RTTI. `std::visit` dispatches on the variant's index, typically through a jump table or, for a few alternatives, a chain of compares; how well it inlines varies between standard library versions. If profiling shows `std::visit` overhead, a switch on `v.index()` with `std::get_if` is a reliable fallback.

An overload set without a catch-all case gives you exhaustiveness checking: adding a new alternative to the variant makes every visit that doesn't handle it fail to compile.

```

// Visitor pattern with lambda
template<typename... Ts>
struct Overload : Ts... {
    using Ts::operator()...;
};

template<typename... Ts>
Overload(Ts...) -> Overload<Ts...>; // needed in C++17, implicit in C++20

// Usage
std::variant<int, double, std::string> value;

std::visit(Overload{
    [](int i) { std::cout << "int: " << i; },
    [](double d) { std::cout << "double: " << d; },
    [](const std::string& s) { std::cout << "string: " << s; }
}, value);

```

Smart Pointers in Low-Latency Code

Smart pointers provide automatic memory management, but their use in low-latency systems requires careful consideration. While `std::unique_ptr` has zero overhead, `std::shared_ptr` involves atomic reference counting that can harm performance. This section explains when and how to use smart pointers without compromising latency.

When to Use Smart Pointers

Smart pointers are excellent for managing ownership and preventing leaks, but should be used judiciously in hot paths. Reserve them for initialization, cleanup, and ownership transfer—avoid them in tight loops where microseconds matter.

```

// DON'T use in hot path
void hot_path_bad() {
    auto order = std::make_shared<Order>(); // Heap allocation (object + control block)
    process(order); // if process takes shared_ptr by value: atomic inc + dec
} // last owner gone: atomic dec, destructor, free

// DO use for ownership management in setup/teardown
class TradingEngine {

```

```

std::unique_ptr<Strategy> strategy_; // Exclusive ownership, no overhead
std::vector<std::unique_ptr<Order>> pending_orders_;

public:
    TradingEngine() {
        strategy_ = std::make_unique<MomentumStrategy>();
    }
    // Automatic cleanup, no manual delete
};

```

std::unique_ptr - Zero Overhead

std::unique_ptr provides exclusive ownership with almost no runtime overhead compared to raw pointers. With the default (or any stateless) deleter it's the same size as a raw pointer, and dereferencing it compiles to the same code.

The one measurable cost: because unique_ptr has a non-trivial destructor, the Itanium C++ ABI (Linux, macOS) passes it by value *in memory* rather than in a register, and the caller must still run the destructor on the moved-from object. For a function called in a tight loop, take T& or T* and keep ownership at the edges (Chandler Carruth, “There Are No Zero-cost Abstractions”, CppCon 2019).

```

// Ownership transfer: cheap, but not literally free
std::unique_ptr<Order> create_order() {
    return std::make_unique<Order>(symbol, price);
}

// Move semantics - no copying
void process_order(std::unique_ptr<Order> order) {
    // Takes ownership
    send_to_exchange(*order);
    // Automatically deleted when function returns
}

// Custom deleter for a pool (the PoolAllocator from Chapter 1):
// destroy() runs ~Order and returns the slot to the pool
struct PoolDeleter {
    void operator()(Order* p) const { order_pool.destroy(p); }
};

std::unique_ptr<Order, PoolDeleter> order(order_pool.create(/* ctor args */));
// Stateless deleter: still pointer-sized

```

Avoiding shared_ptr in Hot Path

std::shared_ptr uses atomic operations for reference counting. Each *copy* incurs an atomic increment and each destruction an atomic decrement: ~20 cycles uncontended, and far more when several cores touch the same control block, which bounces its cache line between them.

```

// BAD: copies the shared_ptr each iteration
void process_orders_bad(const std::vector<std::shared_ptr<Order>>& orders) {
    for (auto order : orders) { // copy: atomic increment
        process(*order);
    } // destruction: atomic decrement
}

// FINE: binding a reference copies nothing, so no refcount traffic

```

```

void process_orders_ok(const std::vector<std::shared_ptr<Order>>& orders) {
    for (const auto& order : orders) {
        process(*order);
    }
}

// The remaining cost is layout: every Order is a separate heap block, so
// iterating chases a pointer to a different cache line each time.

// BETTER: store the objects contiguously
void process_orders_best(const std::vector<Order>& orders) {
    for (const Order& order : orders) {
        process(order); // sequential memory, prefetcher-friendly
    }
}

```

Pass `shared_ptr` by value only when the callee takes shared ownership; otherwise pass `const T&` or `T*`.

Ranges and Views (C++20)

C++20 ranges provide composable, lazy-evaluated operations on sequences. Views are lightweight range adaptors that don't own data and don't copy—perfect for low-latency systems where you need to filter, transform, or process data without allocating intermediate containers.

Range Views for Zero-Copy Operations

Range views enable pipeline-style data processing without creating intermediate containers. All operations are lazily evaluated, meaning work is only done when iterating the final result, eliminating temporary allocations and copies.

```

#include <ranges>

std::vector<Order> orders = get_orders();

// Filter without copying
auto buy_orders = orders
    | std::views::filter([](const Order& o) { return o.is_buy(); })
    | std::views::take(10); // Lazy evaluation, no allocation

// Transform without intermediate storage
auto prices = orders
    | std::views::transform([](const Order& o) { return o.price(); });

// Compose operations
auto top_buy_prices = orders
    | std::views::filter([](const Order& o) { return o.is_buy(); })
    | std::views::transform([](const Order& o) { return o.price(); })
    | std::views::take(5);

// Only computed when iterated
for (double price : top_buy_prices) {
    process(price);
}

```

Custom Ranges

Creating custom range-compatible types enables seamless integration with range algorithms and views. This is particularly useful for circular buffers, custom containers, or memory-mapped data structures common in low-latency systems.

```
// Fixed-capacity buffer that keeps the last N items.
// begin()/end() make it a range; iteration is in storage order, which is
// fine for order-independent work like sums, but not oldest-to-newest once
// the buffer has wrapped (that needs a custom iterator that wraps around).
template<typename T, size_t N>
class RecentItems {
    std::array<T, N> data_{};
    size_t head_ = 0;
    size_t size_ = 0;

public:
    auto begin() const { return data_.begin(); }
    auto end() const { return data_.begin() + size_; }

    void push(const T& item) {
        data_[head_] = item;
        head_ = (head_ + 1) % N;
        if (size_ < N) ++size_;
    }
};

// Use with range algorithms
RecentItems<int, 100> recent;
auto sum = std::ranges::fold_left(recent, 0, std::plus{}); // C++23
// C++20: std::accumulate(recent.begin(), recent.end(), 0)
```

Concepts (C++20)

Concepts provide compile-time constraints on template parameters, enabling clear interface specifications and dramatically better error messages. For low-latency development, concepts help catch errors early and document requirements without any runtime cost.

Type Constraints

Concepts replace cryptic SFINAE with readable constraints. They make template code self-documenting and catch type errors at the point of instantiation, not deep within implementation details.

```
#include <concepts>

// Constraint: T must be integral
template<std::integral T>
T add(T a, T b) {
    return a + b;
}

// Custom concept
template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;
```

```

template<Numeric T>
T multiply(T a, T b) {
    return a * b;
}

// Concept for tradeable instruments. Check a const object, since that's
// how calculate_value uses it: requires(T t) would accept types whose
// price() is non-const, and the call below would then fail to compile
template<typename T>
concept Tradeable = requires(const T& t) {
    { t.symbol() } -> std::convertible_to<std::string>;
    { t.price() } -> std::convertible_to<double>;
    { t.quantity() } -> std::convertible_to<int>;
};

template<Tradeable T>
double calculate_value(const T& instrument) {
    return instrument.price() * instrument.quantity();
}

```

Concepts for Better Error Messages

Without concepts, template errors can produce pages of incomprehensible compiler output. Concepts provide clear, actionable error messages at the point of template instantiation, dramatically reducing debugging time.

```

// Without concepts - the error points deep inside the library
std::list<int> prices = {3, 1, 2};
std::sort(prices.begin(), prices.end());
// error: no match for 'operator-' (operand types are
//   'std::_List_iterator<int>' and 'std::_List_iterator<int>')
//   ... in instantiation of '__introsort_loop' ... (dozens of lines)

// With concepts - the error names the unmet requirement at the call site
std::ranges::sort(prices);
// error: no match for call to 'std::ranges::sort(std::list<int>&)'
// note: the required expression ... 'random_access_range<std::list<int>>'
//       was not satisfied

// Your own templates get the same benefit
template<typename T>
concept Processable = requires(const T& t) {
    { t.process() } -> std::same_as<void>;
};

template<Processable T>
void run(const T& data) {
    data.process();
}
// run(42): error: 'int' does not satisfy 'Processable'

```

Interview Questions & Answers

This section provides comprehensive answers to common interview questions about modern C++ features, with a focus on low-latency implications. Each answer includes practical examples and performance considerations relevant to trading systems.

1. “Explain move semantics and when they improve performance.”

Answer:

Move semantics allow transferring resources from one object to another without copying, introduced in C++11.

Key Concepts:

Rvalue References:

```
void process(Order&& order) { // Rvalue reference
    // 'order' is about to be destroyed, safe to steal its resources
}

Order create() {
    Order order;
    return order; // NRVO, or implicitly moved: a returned local is treated as an rvalue
}

process(create()); // create() is a prvalue: binds to Order&&, no copy
```

Move vs Copy:

```
class Buffer {
    char* data_;
    size_t size_;

public:
    // Copy: Allocate + deep copy
    Buffer(const Buffer& other) : size_(other.size_) {
        data_ = new char[size_];
        std::memcpy(data_, other.data_, size_); // Expensive!
    }

    // Move: Just steal pointer
    Buffer(Buffer&& other) noexcept
        : data_(other.data_), size_(other.size_) {
        other.data_ = nullptr;
        other.size_ = 0;
        // Just pointer reassignment - cheap!
    }
};
```

When Performance Improves:

1. Returning from functions:

```
std::vector<Order> create_orders() {
    std::vector<Order> orders;
    // ... populate ...
    return orders; // NRVO, or a move at worst - never a copy
}
```

2. Container operations:

```
std::vector<Order> orders;
orders.push_back(create_order()); // Move into vector
```

```
Order temp = create_order();
orders.push_back(std::move(temp)); // Explicit move
```

3. Swapping:

```
// Before C++11, std::swap did 3 copies. Since C++11 it does exactly this:
Order temp = std::move(a);
a = std::move(b);
b = std::move(temp);
// 3 moves (much faster). Just call std::swap(a, b).
```

4. **Growing a vector:** reallocation moves the elements instead of copying them, but only if the move constructor is `noexcept`.

Performance Impact:

- Copy a 1MB string: ~50-100 s (allocation, page faults on fresh memory, and a memcpy at ~10-20 GB/s)
- Move a 1MB string: a few ns (copy a pointer and two sizes, null out the source)
- **Thousands of times faster**, and the gap grows with the size of the data

Best Practices:

- Always mark move constructors `noexcept` (or let the compiler generate them)
- Use `std::move` when you're done with an object
- Return by value, trust RVO/move semantics; don't write `return std::move(local);`, which disables NRVO
- `std::move` on a `const` object compiles but silently *copies*: the result is `const T&&`, which only the copy constructor accepts. Don't declare objects `const` if you intend to move from them.

2. “What is perfect forwarding and why is it useful?”

Answer:

Perfect forwarding preserves the value category (lvalue/rvalue) of arguments when passing them through template functions.

The Problem:

```
template<typename T>
void wrapper(T arg) {
    function(arg); // 'arg' is always an lvalue here!
    // Even if we called wrapper with an rvalue
}
```

```
wrapper(create_order()); // Temporary (rvalue) becomes lvalue in wrapper
// Move semantics lost!
```

The Solution:

```
template<typename T>
void wrapper(T&& arg) { // Universal/forwarding reference
    function(std::forward<T>(arg)); // Preserves value category
}

// If called with rvalue:
```

```
wrapper(create_order()); // T deduced as Order, arg is Order&&
```

```
// If called with lvalue:
```

```
Order order;
```

```
wrapper(order); // T deduced as Order&, arg is Order&
```

How It Works:

Reference Collapsing Rules:

```
using LRef = Order&;
```

```
using RRef = Order&&;
```

```
LRef& → Order& // && & → &
```

```
LRef&& → Order& // && && → &
```

```
RRef& → Order& // && & → &
```

```
RRef&& → Order&& // && && → &&
```

std::forward Implementation (simplified):

```
template<typename T>
```

```
T&& forward(std::remove_reference_t<T>& arg) {
```

```
    return static_cast<T&&>(arg);
```

```
}
```

```
// If T is Order (rvalue), returns Order&&
```

```
// If T is Order& (lvalue), returns Order& && → Order&
```

Real-World Example:

```
// Factory function
```

```
template<typename T, typename... Args>
```

```
std::unique_ptr<T> make_unique(Args&&... args) {
```

```
    return std::unique_ptr<T>(
        new T(std::forward<Args>(args)...) // Perfect forward to constructor
    );
```

```
}
```

```
// Usage
```

```
auto order = make_unique<Order>(symbol, price);
```

```
// If symbol is temporary, moves into Order
```

```
// If symbol is lvalue, copies into Order
```

Key Interview Points:

- Forwarding references: T&& in template context
- std::forward preserves lvalue/rvalue-ness
- Essential for wrapper functions and factories
- Enables zero-overhead abstraction

3. “Explain constexpr and its benefits for low-latency code.”

Answer:

constexpr allows computation at compile time, moving work from runtime to compilation.

Basic Usage:

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
```

```

}

// Computed at compile time
constexpr int result = factorial(5); // 120
std::array<int, factorial(5)> buffer; // Size = 120

// Can still be used at runtime
int runtime_n = get_input();
int runtime_result = factorial(runtime_n); // Computed at runtime

```

Benefits for Low-Latency:

1. Guaranteed Compile-Time Evaluation:

The optimizer already folds simple expressions like `64 * 1024` without any help. What `constexpr` adds is a *guarantee*: a `constexpr` variable must be computed at compile time (or the build fails), and the result can be used where the language requires a constant, like array sizes, template arguments and `static_assert`.

```

// Without constexpr: the compiler MAY fold this, but can't use it
// as a constant expression
int get_buffer_size() {
    return 64 * 1024;
}
// char buffer[get_buffer_size()]; // Error: not a constant expression

// With constexpr - usable as a compile-time constant
constexpr int get_buffer_size_ce() {
    return 64 * 1024;
}

constexpr int SIZE = get_buffer_size_ce(); // 65536, guaranteed at compile time
char buffer[SIZE];

// C++20 consteval: must ALWAYS be evaluated at compile time
constexpr int must_be_compile_time(int x) { return x * x; }

```

2. Lookup Tables:

```

// Generate CRC table at compile time
constexpr auto generate_crc_table() {
    std::array<uint32_t, 256> table{};
    for (uint32_t i = 0; i < 256; ++i) {
        uint32_t crc = i;
        for (int j = 0; j < 8; ++j) {
            crc = (crc >> 1) ^ ((crc & 1) ? 0xEDB88320 : 0);
        }
        table[i] = crc;
    }
    return table;
}

constexpr auto CRC_TABLE = generate_crc_table();

// Runtime: one table lookup per byte instead of 8 shift/xor steps
uint32_t crc32(const unsigned char* data, size_t len) {
    uint32_t crc = 0xFFFFFFFF;
    for (size_t i = 0; i < len; ++i) {

```

```

        crc = CRC_TABLE[(crc ^ data[i]) & 0xFF] ^ (crc >> 8);
    }
    return ~crc;
}
// (For CRC32C, x86's SSE4.2 crc32 instruction is faster still)

```

3. Configuration Validation:

```

struct Config {
    int max_orders;
    int max_queue_size;

    constexpr Config(int orders, int queue)
        : max_orders(orders), max_queue_size(queue) {
        if (queue < orders) {
            throw "Queue too small!"; // Compile error!
        }
    }
};

constexpr Config cfg{1000, 2000}; // OK
// constexpr Config bad{1000, 500}; // Compile error!

```

4. Type Computation:

```

template<size_t N>
constexpr auto optimal_int_type() {
    if constexpr (N <= 255) {
        return uint8_t{};
    } else if constexpr (N <= 65535) {
        return uint16_t{};
    } else {
        return uint32_t{};
    }
}

// Choose smallest integer type at compile time
using OrderID = decltype(optimal_int_type<MAX_ORDERS>());

```

Modern Extensions (C++20):

```

// constexpr algorithms
constexpr auto find_max() {
    std::array data = {1, 5, 3, 9, 2};
    return *std::max_element(data.begin(), data.end());
}

constexpr int MAX_VAL = find_max(); // 9, compile-time

// constexpr virtual functions
struct Base {
    constexpr virtual int value() const = 0;
};

struct Derived : Base {
    constexpr int value() const override { return 42; }
};

```

Performance Impact:

- Latency: 0ns for the computation itself (done before the program runs); a table still costs memory accesses
- Binary size: Potentially larger (tables embedded)
- Startup time: Faster (no initialization needed), and no static-initialization-order problems

Key Interview Points:

- Moves work from runtime to compile time, with a guarantee rather than a hope
- Useful for lookup tables, but only where a lookup beats computing
- Validates configuration at compile time
- Works with most C++ features in C++20+

4. “When would you use `std::variant` over inheritance?”

Answer:

`std::variant` provides a type-safe union with value semantics, while inheritance uses polymorphism with reference/pointer semantics.

Comparison:

Inheritance Approach:

```
class Order {
public:
    virtual ~Order() = default;
    virtual void execute() = 0;
    virtual double get_price() const = 0;
};

class MarketOrder : public Order {
    double quantity_;
public:
    void execute() override { /* ... */ }
    double get_price() const override { return market_price(); }
};

class LimitOrder : public Order {
    double price_;
    double quantity_;
public:
    void execute() override { /* ... */ }
    double get_price() const override { return price_; }
};

// Holding different order types in one container requires pointers,
// which in practice means one heap allocation per object
std::unique_ptr<Order> order = std::make_unique<MarketOrder>();
order->execute(); // Virtual dispatch
```

`std::variant` Approach:

```
struct MarketOrder {
    double quantity;
    void execute() const { /* ... */ }
    double get_price() const { return market_price(); }
};
```

```

struct LimitOrder {
    double price;
    double quantity;
    void execute() const { /* ... */ }
    double get_price() const { return price; }
};

using Order = std::variant<MarketOrder, LimitOrder>;

// Usage - value semantics
Order order = MarketOrder{100.0};
std::visit([](auto&& o) { o.execute(); }, order); // No virtual dispatch

```

When to use `std::variant`:

1. Performance-Critical Code:

```

// std::variant: No heap allocation, no virtual dispatch
// Size = max(sizeof(MarketOrder), sizeof(LimitOrder)) + discriminator

// Inheritance: Requires heap, vtable lookup, cache misses

```

2. Value Semantics:

```

std::vector<Order> orders; // Contiguous storage
orders.push_back(MarketOrder{100.0});
orders.push_back(LimitOrder{150.0, 50.0});

// vs inheritance
std::vector<std::unique_ptr<Order>> orders; // Pointers scattered in heap

```

3. Closed Set of Types:

```

// Variant: All types known at compile time
using Message = std::variant<Heartbeat, NewOrder, CancelOrder>;

// One lambda per alternative, no catch-all: if a new message type is
// added to Message, this stops compiling until it's handled
std::visit(Overload{
    [](const Heartbeat& hb) { /* ... */ },
    [](const NewOrder& no) { /* ... */ },
    [](const CancelOrder& co) { /* ... */ },
}, message);

// By contrast, an `if constexpr` chain ending in a plain `else` (or a
// generic `auto` lambda) silently accepts new types: no exhaustiveness check

```

Performance Comparison:

```

// Inheritance: each element is a pointer to a separate heap object
for (auto& order : orders) { // std::vector<std::unique_ptr<Order>>
    order->execute(); // load vptr, load target, indirect call
}

// std::variant: elements stored inline, contiguously
for (auto& order : orders) { // std::vector<std::variant<...>>
    std::visit([](auto&& o) { o.execute(); }, order); // index-based dispatch
}

```

```
}
```

Don't quote a fixed speed-up; it depends on the workload. A well-predicted virtual call costs only a few cycles. The variant version usually wins for two reasons: the objects are contiguous, so there's no cache miss per element, and the compiler can often inline each alternative's code. When the objects are hot in cache and the calls are predictable, the difference can be small. Benchmark your own case.

When to use Inheritance:

1. **Open set of types** (users can extend)
2. **Runtime polymorphism required**
3. **Type erasure needed**
4. **Large type hierarchies**

std::variant Advantages:

- Value semantics (copyable, movable)
- No heap allocation
- No virtual dispatch overhead
- Cache-friendly (contiguous storage)
- Compile-time exhaustiveness checking (with an overload-set visitor)
- Usable in constant expressions (C++20)

std::variant Disadvantages:

- Size = largest alternative
- Closed set of types
- More verbose visitor syntax
- Can't add types without recompiling

Real-World Low-Latency Example:

```
// Message protocol with variant
using MarketDataMsg = std::variant<
    QuoteTick,
    TradeTick,
    OrderBookUpdate
>;

void process_message(const MarketDataMsg& msg) {
    std::visit([](const auto& data) {
        // Inlined, no virtual dispatch
        // All code in one translation unit
        // Perfect for optimization
        process_impl(data);
    }, msg);
}
```

Key Interview Points:

- Use variant for closed, performance-critical type sets
- Use inheritance for extensible, general-purpose designs
- Variant enables better optimization and cache locality
- Both have their place - choose based on requirements

5. “How do you use templates to achieve zero-cost abstractions?”

Answer:

Zero-cost abstractions mean you pay no runtime cost for the abstraction - the generated code is as efficient as hand-written code.

Core Principle:

“What you don’t use, you don’t pay for. What you do use, you couldn’t hand-code any better.” - Bjarne Stroustrup

Technique 1: Compile-Time Polymorphism

```
// Runtime polymorphism (virtual functions)
class Sorter {
public:
    virtual void sort(int* data, size_t size) = 0; // vtable lookup
};

class QuickSort : public Sorter {
    void sort(int* data, size_t size) override { /* ... */ }
};

void process(Sorter& sorter, int* data, size_t size) {
    sorter.sort(data, size); // Virtual call - runtime cost
}

// Compile-time polymorphism (templates)
template<typename Sorter>
void process(Sorter& sorter, int* data, size_t size) {
    sorter.sort(data, size); // Inlined - zero cost!
}

class QuickSort {
public:
    void sort(int* data, size_t size) { /* ... */ } // No virtual
};

// Instantiates optimized version for QuickSort
QuickSort sorter;
process(sorter, data, size); // Fully inlined, no vtable
```

Technique 2: Type-Based Optimization

```
template<typename T>
void copy_data(T* dest, const T* src, size_t count) {
    if constexpr (std::is_trivially_copyable_v<T>) {
        // Trivially copyable: one bulk memcpy (vectorized copy loop or
        // `rep movsb`) instead of an element-by-element loop
        std::memcpy(dest, src, count * sizeof(T));
    } else {
        // Complex types: call copy constructor
        for (size_t i = 0; i < count; ++i) {
            new (&dest[i]) T(src[i]);
        }
    }
}

// For int: compiles to memcpy
copy_data(int_dest, int_src, 1000);
```

```
// For std::string: compiles to loop with copy constructors
copy_data(str_dest, str_src, 1000);
```

```
// No runtime check - branch eliminated at compile time!
```

Technique 3: Policy-Based Design

Policy-based design allows you to inject compile-time behavior through template parameters. Different policies generate completely different code paths, all resolved at compile time with zero runtime overhead.

```
// Different locking strategies, chosen at compile time
struct NoLock { // single-threaded use: compiles to nothing
    void lock() {}
    void unlock() {}
};

struct SpinLockPolicy { // shared use on dedicated cores
    std::atomic<bool> locked{false};
    void lock() { while (locked.exchange(true, std::memory_order_acquire)) cpu_relax(); }
    void unlock() { locked.store(false, std::memory_order_release); }
};

template<typename T, size_t N, typename LockPolicy = NoLock>
class ObjectPool {
    LockPolicy lock_;
    PoolAllocator<T, N> pool_; // from Chapter 1

public:
    T* acquire() {
        std::lock_guard guard(lock_); // NoLock: the calls inline away
        return pool_.create();
    }
    void release(T* obj) {
        std::lock_guard guard(lock_);
        pool_.destroy(obj);
    }
};

// Owned by one thread: no synchronization cost at all
ObjectPool<Order, 1024> local_pool;

// Shared between threads: same code, with a lock
ObjectPool<Order, 1024, SpinLockPolicy> shared_pool;

// No runtime overhead - the policy type determines behavior at compile time
```

Technique 4: Expression Templates

Expression templates defer computation and eliminate temporary objects in complex expressions. Instead of evaluating each operation immediately, they build an expression tree at compile time that evaluates in a single pass.

```
// Avoid temporary vectors in arithmetic (C++20)
template<typename E>
concept VecExpr = requires(const E& e, size_t i) {
    { e[i] } -> std::convertible_to<double>;
};
```

```

    { e.size() } -> std::convertible_to<size_t>;
};

// a + b as a lightweight object that computes elements on demand
template<VecExpr L, VecExpr R>
class VecSum {
    const L& lhs_; // references: the expression must not outlive its operands
    const R& rhs_;
public:
    VecSum(const L& l, const R& r) : lhs_(l), rhs_(r) {}
    double operator[](size_t i) const { return lhs_[i] + rhs_[i]; } // Lazy
    size_t size() const { return lhs_.size(); }
};

template<VecExpr L, VecExpr R>
VecSum<L, R> operator+(const L& l, const R& r) { return {l, r}; }

class Vec {
    std::vector<double> data_;
public:
    explicit Vec(size_t n) : data_(n) {}

    // Evaluating an expression: the single loop happens here
    template<VecExpr E>
    Vec(const E& expr) : data_(expr.size()) {
        for (size_t i = 0; i < data_.size(); ++i) data_[i] = expr[i];
    }

    double operator[](size_t i) const { return data_[i]; }
    size_t size() const { return data_.size(); }
};

// Usage
Vec a(1000), b(1000), c(1000);
Vec result = a + b + c; // No temporary Vecs!
// Compiles to a single loop: result[i] = a[i] + b[i] + c[i]

// Pitfall: `auto expr = a + b + c;` stores references to the temporary
// VecSum for (a + b), which dies at the end of the statement → dangling.
// Libraries like Eigen document the same trap.

```

Technique 5: CRTP (Curiously Recurring Template Pattern)

CRTP enables static polymorphism by having a derived class pass itself as a template parameter to its base. This allows the base class to call derived methods through static casts, achieving polymorphism without virtual functions.

```

// Static polymorphism without virtual functions
template<typename Derived>
class MessageHandler {
public:
    void handle(const Message& msg) {
        static_cast<Derived*>(this)->handle_impl(msg);
        // Resolved at compile time - inlined!
    }
}

```

```

};

class OrderHandler : public MessageHandler<OrderHandler> {
public:
    void handle_impl(const Message& msg) {
        // Actual implementation
    }
};

template<typename Handler>
void process_messages(Handler& handler, const std::vector<Message>& msgs) {
    for (const auto& msg : msgs) {
        handler.handle(msg); // Inlined, no virtual call
    }
}

```

Technique 6: Tag Dispatch

Tag dispatch uses empty tag types and ordinary overload resolution to select algorithm implementations at compile time based on type properties. It is how the STL selects algorithms by iterator category, and it predates `if constexpr`.

```

// Compile-time algorithm selection by iterator category
// (named my_advance to avoid clashing with std::advance via ADL)
template<typename Iterator>
void advance_impl(Iterator& it, int n, std::random_access_iterator_tag) {
    it += n; // Random access: O(1)
}

template<typename Iterator>
void advance_impl(Iterator& it, int n, std::input_iterator_tag) {
    for (int i = 0; i < n; ++i) {
        ++it; // Forward-only: O(n); n must be non-negative
    }
}

template<typename Iterator>
void my_advance(Iterator& it, int n) {
    // The category object picks the overload. Tags inherit from each other
    // (forward_iterator_tag derives from input_iterator_tag), so a forward
    // iterator's tag matches the input_iterator_tag overload.
    advance_impl(it, n,
        typename std::iterator_traits<Iterator>::iterator_category{});
}

// Since C++17, `if constexpr` usually replaces tag dispatch:
template<typename Iterator>
void my_advance17(Iterator& it, int n) {
    if constexpr (std::random_access_iterator<Iterator>) { // C++20 concept
        it += n;
    } else {
        while (n-- > 0) ++it;
    }
}

```

Real-World Example: Matrix Operations

```

template<typename T, size_t Rows, size_t Cols>
class Matrix {
    std::array<std::array<T, Cols>, Rows> data_;

public:
    // Size known at compile time
    static constexpr size_t rows() { return Rows; }
    static constexpr size_t cols() { return Cols; }

    // Element access inlined (const overload needed for `rhs(k, j)` below)
    T& operator()(size_t r, size_t c) {
        return data_[r][c];
    }
    const T& operator()(size_t r, size_t c) const {
        return data_[r][c];
    }

    // Multiplication optimized per size
    template<size_t OtherCols>
    Matrix<T, Rows, OtherCols> operator*(
        const Matrix<T, Cols, OtherCols>& rhs) const {

        Matrix<T, Rows, OtherCols> result{};

        // Loop bounds known at compile time
        // Compiler can unroll, vectorize, reorder
        for (size_t i = 0; i < Rows; ++i) {
            for (size_t k = 0; k < Cols; ++k) {
                for (size_t j = 0; j < OtherCols; ++j) {
                    result(i, j) += data_[i][k] * rhs(k, j);
                }
            }
        }
        return result;
    }
};

// Different code generated for each size
Matrix<double, 3, 3> a, b;
auto c = a * b; // Fully unrolled, vectorized 3x3 multiply

Matrix<double, 4, 4> d, e;
auto f = d * e; // Different optimized code for 4x4

// No runtime checks for dimensions
// No virtual functions
// Perfect compiler optimization

```

Performance Comparison:

```

// Virtual functions
class Base { virtual void compute() = 0; };
// Assembly: load vptr, then call [vptr + offset] - an indirect call.
// Predicted correctly it costs only a few cycles; mispredicted, ~15-20.
// The bigger cost: the compiler can't inline through it.

```

```
// Template
template<typename T>
void compute(T& obj) { obj.compute(); }
// Assembly: direct call, or fully inlined and optimized with the caller
```

Key Interview Points:

- Templates enable compile-time decisions
 - No runtime cost: branches eliminated, calls inlined
 - Compiler sees full picture and optimizes aggressively
 - Type information drives optimization
 - Trade-off: Longer compile times, larger binary size
 - Perfect for hot paths in low-latency systems
-

Summary

Modern C++ features enable writing expressive, safe code without sacrificing performance. The key to effective low-latency C++ is understanding which abstractions compile away completely and which introduce runtime cost.

Core Principles:

- **Move semantics:** Eliminate unnecessary copies through resource transfer
- **Perfect forwarding:** Zero-overhead argument forwarding in generic code
- **constexpr:** Move computation from runtime to compile time
- **Templates:** Enable static polymorphism and zero-cost abstractions
- **std::variant/optional:** Type-safe alternatives with value semantics
- **Ranges:** Lazy evaluation and composition without temporary allocations
- **Concepts:** Better compile-time errors with no runtime cost

Low-Latency Guidelines:

1. Prefer compile-time to runtime computation
2. Use value semantics when possible (avoid heap allocation)
3. Leverage move semantics to eliminate copies
4. Choose **std::variant** over virtual functions in hot paths when the set of types is closed
5. Reserve smart pointers for ownership management, not hot paths
6. Use ranges for allocation-free data processing pipelines (and check the generated code: some adaptors, like **filter**, add overhead)
7. Apply concepts for clear interfaces without overhead

The trade-off is between developer time (compile time, code complexity) and runtime performance. In low-latency systems, this trade-off almost always favors runtime performance—making these modern C++ features essential tools in the high-frequency trading developer’s toolkit.

Contents

Chapter 4: Low-Latency Techniques & Best Practices	1
Table of Contents	1
Busy-Waiting vs Sleeping	2
Understanding the Trade-off	2
CPU Pause Instruction	2
Kernel Bypass & User-Space Networking	2
Traditional Network Stack (Slow)	2
Kernel Bypass (Fast)	3
DPDK (Data Plane Development Kit)	3
OpenOnload and ef_vi (AMD, formerly Solarflare)	4
How Polling a NIC Works	4
Minimizing System Calls	5
System Call Overhead	5
Batching System Calls	5
Avoiding getpid(), gettid()	6
Async I/O (io_uring)	6
High-Resolution Timing	7
TSC (Time Stamp Counter)	7
HPET (High Precision Event Timer)	8
Hardware Timestamping	8
TCP/UDP Performance Optimization	10
TCP Socket Tuning	10
Zero-Copy Send (MSG_ZEROCOPY): Usually Not for Trading	10
Multicast Optimization	10
SIMD and Vectorization	11
Manual SIMD with Intrinsics	11
Auto-Vectorization	12
Profiling and Measurement	12
CPU Performance Counters (perf)	12
Intel VTune	13
Manual Instrumentation	13
Latency Histogram	13
Interview Questions & Answers	13
1. “How do you measure and optimize tail latency (p99, p99.9)?”	13
2. “Explain kernel bypass and when you would use it.”	17
3. “How do you handle packet loss and retransmission in a low-latency system?”	18
4. “Explain how SIMD can improve performance and give an example.”	22
5. “How would you debug a latency spike in production?”	24
Summary	28

Chapter 4: Low-Latency Techniques & Best Practices

Table of Contents

1. Busy-Waiting vs Sleeping
2. Kernel Bypass & User-Space Networking
3. Minimizing System Calls
4. High-Resolution Timing
5. TCP/UDP Performance Optimization
6. SIMD and Vectorization
7. Profiling and Measurement
8. Interview Questions & Answers

Busy-Waiting vs Sleeping

Understanding the Trade-off

Chapter 2 covers the mechanics: sleeping vs spinning vs hybrid waits and the PAUSE instruction. The numbers to remember:

Strategy	Reaction time	CPU cost
<code>sleep_for(1µs)</code>	~50-60µs (Linux timer slack + wake-up)	Near zero
Blocking on a futex/condition variable	~2-10µs (wake-up + context switch)	Near zero
Spin with <code>_mm_pause()</code>	~40ns (one PAUSE on Skylake+) + cache-line transfer	One full core
Spin without pause	Tens of ns, plus a pipeline flush on exit	One full core, and slows the hyperthread sibling

For the latency-critical path on a dedicated core, spin. Everywhere else, block.

CPU Pause Instruction

`cpu_relax()` (defined in Chapter 2) wraps `_mm_pause()`. Under contention, waiting longer between checks reduces cache-line traffic; this matters when several threads spin on the same variable, not for a single dedicated poller:

```
// Exponential backoff
void wait_with_backoff(std::atomic<bool>& ready) {
    int delay = 1;
    while (!ready.load(std::memory_order_acquire)) {
        for (int i = 0; i < delay; ++i) {
            cpu_relax();
        }
        delay = std::min(delay * 2, 1024); // Cap at 1024
    }
}
```

Kernel Bypass & User-Space Networking

Traditional Network Stack (Slow)

```
Application
  ↓ system call (user → kernel mode switch)
Kernel Space
  ↓ copy between user buffer and socket buffer
TCP/IP Stack
  ↓
Network Driver
  ↓ DMA
NIC
```

Receive side: interrupt → softirq → wake the blocked thread
Total latency: ~5-20 s per packet

Mode switches: 1+ per send/receive; context switch too if the receiver was asleep
Memory copies: 1 each way (user kernel)

Kernel Bypass (Fast)

Application (User Space)

↓ writes descriptors / polls completions in memory shared with the NIC

NIC Ring Buffer

↓ DMA

NIC

Total latency: ~0.5-2 s per packet

Mode switches: 0 on the fast path

Memory copies: 0-1 (depends on the API: zero-copy APIs vs socket emulation)

DPDK (Data Plane Development Kit)

```
#include <rte_eal.h>
#include <rte_ethdev.h>
#include <rte_mbuf.h>

class DPDKReceiver {
    const uint16_t port_id_ = 0;
    const uint16_t queue_id_ = 0;

    struct rte_mempool* mbuf_pool_;

public:
    void init() {
        // Initialize DPDK (EAL args: cores 0-3, 4 memory channels)
        const char* argv[] = {"app", "-l", "0-3", "-n", "4"};
        int argc = sizeof(argv) / sizeof(argv[0]);
        if (rte_eal_init(argc, const_cast<char**>(argv)) < 0) {
            throw std::runtime_error("EAL init failed");
        }

        // Create packet buffer pool
        mbuf_pool_ = rte_pktmbuf_pool_create(
            "MBUF_POOL",
            8192,           // Number of buffers
            256,           // Cache size
            0,
            RTE_MBUF_DEFAULT_BUF_SIZE,
            rte_socket_id()
        );

        // Configure port: 1 RX queue, 1 TX queue, default settings
        struct rte_eth_conf port_conf = {};
        rte_eth_dev_configure(port_id_, 1, 1, &port_conf);
        // (TX queue setup and error checks omitted for brevity)
        rte_eth_rx_queue_setup(port_id_, queue_id_, 1024,
                               rte_socket_id(), NULL, mbuf_pool_);
        rte_eth_dev_start(port_id_);
    }
}
```

```

void receive_loop() {
    struct rte_mbuf* bufs[32];

    while (true) {
        // Poll for packets (no system calls!)
        uint16_t nb_rx = rte_eth_rx_burst(
            port_id_, queue_id_, bufs, 32
        );

        for (uint16_t i = 0; i < nb_rx; ++i) {
            // The NIC has DMA'd the packet into this mbuf in host
            // memory; process it in place, no copy
            process_packet(bufs[i]);
        }
        rte_pktmbuf_free_bulk(bufs, nb_rx); // return buffers to the pool
    }
};

```

DPDK has no TCP/IP stack of its own, so in trading it's mostly used for UDP market data. Order entry over TCP typically uses a vendor stack instead.

OpenOnload and ef_vi (AMD, formerly Solarflare)

```

// Transparent kernel bypass (LD_PRELOAD)
// No code changes needed!

// Just run with:
// onload --profile=latency ./trading_app

// Provides:
// - A user-space TCP/UDP stack behind the normal socket API
// - Hardware timestamping
// - ~1-2µs socket latency instead of ~5-20µs

// For the lowest latency, ef_vi is the same vendor's raw packet API
// (no TCP/IP stack, similar in spirit to DPDK)

```

Other common options: NVIDIA/Mellanox VMA (socket acceleration, like Onload) and Cisco (formerly Exablaze) ExaNIC's libexanic.

How Polling a NIC Works

This is illustrative, not a real driver: real NICs are driven through vendor libraries (DPDK drivers, ef_vi, libexanic). The key design point is that the hot path polls a descriptor ring in *host* memory that the NIC writes by DMA. Reading a NIC register over PCIe (MMIO) costs ~0.5-1µs, so drivers avoid MMIO reads on the fast path and only do posted MMIO writes ("doorbells").

```

struct RxDescriptor { // written by the NIC via DMA
    uint64_t buffer_addr;
    uint16_t length;
    volatile uint8_t done; // NIC sets this when the packet is complete
};

class PollingRxRing {
    RxDescriptor* ring_; // host memory shared with the NIC
};

```

```

uint8_t* buffers_;           // packet buffers, also DMA targets
volatile uint32_t* doorbell_; // NIC register mapped with mmap (MMIO)
uint32_t next_ = 0;

public:
void poll() {
    RxDescriptor& d = ring_[next_];
    if (!d.done) return; // cheap: a cache hit until the NIC's DMA write

    // Read the descriptor before the payload. Real drivers use a
    // platform barrier here (e.g. rte_rmb()); on x86 loads aren't
    // reordered with other loads, so a compiler barrier suffices.
    std::atomic_signal_fence(std::memory_order_acquire);

    process_packet(buffers_ + next_ * BUF_SIZE, d.length);

    d.done = 0;
    next_ = (next_ + 1) % RING_SIZE;
    *doorbell_ = next_; // posted MMIO write: hand the slot back to the NIC
}
};

```

Minimizing System Calls

System Call Overhead

A trivial system call costs ~100ns on current Linux, and 200-500ns or more with Spectre/Meltdown mitigations (KPTI etc.) enabled, before any work is done. It also pollutes caches and TLB.

Not everything that looks like a system call is one:

```

// vDSO: runs in user space, no kernel entry (when the clocksource is TSC)
auto now = std::chrono::system_clock::now(); // clock_gettime: ~20ns
auto t    = std::chrono::steady_clock::now(); // same mechanism

// Raw CPU instruction: ~6-8ns (~25 cycles)
uint64_t tsc = __rdtsc();

```

If clock_gettime suddenly costs ~1µs, the kernel has fallen back from TSC to another clocksource (check /sys/devices/system/clocksource/clocksource0/current_clocksource).

Batching System Calls

```

// BAD: One system call per message
for (const auto& msg : messages) {
    write(socket_fd, msg.data(), msg.size()); // N system calls
}

// GOOD: Gather scattered buffers into one system call
std::vector<struct iovec> iov; // reserve()d once, reused
iov.clear();
for (const auto& msg : messages) {
    iov.push_back({const_cast<char*>(msg.data()), msg.size()});
}
// At most IOV_MAX (1024 on Linux) entries per call; on a TCP socket,

```

```
// check the return value for a partial write
ssize_t n = writev(socket_fd, iov.data(), static_cast<int>(iov.size()));

// If the data is already contiguous (e.g. a std::vector of fixed-size
// structs), a single write() of the whole block does the same job

// UDP: sendmmsg() sends many datagrams in one call; recvmmsg() receives many
```

Batching trades latency for throughput: the first message waits for the batch. In an order path you send immediately; batching is for things like logging and drop copies.

Avoiding getpid(), gettid()

```
// BAD: System call every time
void log_message(const char* msg) {
    int tid = syscall(SYS_gettid); // System call
    printf("[%d] %s\n", tid, msg);
}

// GOOD: Cache thread ID
thread_local int cached_tid = syscall(SYS_gettid); // Once per thread

void log_message(const char* msg) {
    printf("[%d] %s\n", cached_tid, msg); // No gettid system call
}
```

The `printf` is still the bigger problem: formatting is slow and the eventual `write` is a system call. Hot-path logging should copy the raw arguments into a lock-free queue and let a background thread format and write them (e.g. Quill, NanoLog).

Async I/O (io_uring)

```
#include <liburing.h>

class AsyncIO {
    struct io_uring ring_;

public:
    void init() {
        // IORING_SETUP_SQPOLL: a kernel thread polls the submission queue,
        // so submitting needs no system call at all (costs a core)
        struct io_uring_params params = {};
        params.flags = IORING_SETUP_SQPOLL;
        params.sq_thread_idle = 2000; // ms before the poller sleeps
        io_uring_queue_init_params(256, &ring_, &params);
    }

    // Queue a read; nothing reaches the kernel until flush()
    bool queue_read(int fd, void* buf, size_t len, uint64_t offset) {
        struct io_uring_sqe* sqe = io_uring_get_sqe(&ring_);
        if (!sqe) return false; // submission queue full: flush() first

        io_uring_prep_read(sqe, fd, buf, len, offset);
        io_uring_sqe_set_data(sqe, buf);
        return true;
    }
}
```

```

// One submit for everything queued since the last call
void flush() {
    io_uring_submit(&ring_);
}

void poll_completions() {
    struct io_uring_cqe* cqe;

    // Poll for completed operations (no system call)
    while (io_uring_peek_cqe(&ring_, &cqe) == 0) {
        void* buf = io_uring_cqe_get_data(cqe);
        int bytes_read = cqe->res; // negative = -errno

        process_data(buf, bytes_read);

        io_uring_cqe_seen(&ring_, cqe);
    }
};

```

In trading systems `io_uring` is mostly used for disk and logging I/O; the network hot path uses kernel bypass.

High-Resolution Timing

TSC (Time Stamp Counter)

Requirements: the CPU must have an *invariant* TSC (`constant_tsc` and `nonstop_tsc` in `/proc/cpuinfo`), which ticks at a fixed rate regardless of frequency scaling and sleep states. All modern x86 server CPUs do, and on a single socket the TSCs are synchronized across cores.

```

#include <x86intrin.h>
#include <chrono>
#include <thread>

class TSCClock {
    double ns_per_tick_ = 0;

public:
    TSCClock() { calibrate(); } // takes ~100ms: create once at startup

    void calibrate() {
        // Measure TSC frequency against a monotonic clock
        // (high_resolution_clock may be system_clock, which can jump)
        using clock = std::chrono::steady_clock;
        auto t1 = clock::now();
        uint64_t tsc1 = __rdtsc();

        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        uint64_t tsc2 = __rdtsc();
        auto t2 = clock::now();

        auto duration_ns = std::chrono::duration_cast<

```

```

        std::chrono::nanoseconds>(t2 - t1).count();

        // double, not an integer: 2.9 ticks/ns truncated to 2 would be a 45% error
        ns_per_tick_ = static_cast<double>(duration_ns) / (tsc2 - tsc1);
    }

    // Hot path: just read the counter (~25 cycles)
    static uint64_t ticks() { return __rdtsc(); }

    // Off the hot path: convert a tick difference to nanoseconds
    double to_ns(uint64_t tick_delta) const { return tick_delta * ns_per_tick_; }

    // Ordered read, for timing short code sections precisely.
    // RDTSC alone can execute before earlier instructions finish.
    // RDTSCP waits for them, and the LFENCE stops later instructions
    // from starting before the timestamp is taken.
    static uint64_t ticks_ordered() {
        unsigned int aux;
        uint64_t t = __rdtscp(&aux);
        _mm_lfence();
        return t;
    }
};

// Usage: one clock for the whole process
static const TSCClock tsc_clock;

uint64_t start = TSCClock::ticks();
process_order();
uint64_t end = TSCClock::ticks();
record_latency_ticks(end - start); // convert with to_ns() when reporting

```

A 100ms calibration is accurate to roughly 0.01%. For better accuracy, read the kernel's own calibration (tsc: Refined TSC clocksource calibration in dmesg) or CPUID leaf 0x15 where available.

HPET (High Precision Event Timer)

HPET is a hardware timer on the chipset, and it is *not* suitable for low-latency timestamps: each read is an MMIO access that takes on the order of a microsecond, orders of magnitude slower than RDTSC. Linux only uses it as a fallback clocksource. If `current_clocksource` shows `hpet`, every `clock_gettime` becomes that slow; the usual fix is making the TSC usable again (BIOS settings, `tsc=reliable` on the kernel command line if you know it is).

Hardware Timestamping

```

#include <linux/net_tstamp.h>
#include <linux/sockios.h>

class HardwareTimestamps {
    int sock_fd_; // an open UDP socket bound to the interface

public:
    void enable_hw_timestamps() {
        struct ifreq ifr = {};
        std::strncpy(ifr.ifr_name, "eth0", IFNAMSIZ - 1);
    }
};

```

```

    struct hwtstamp_config config = {};
    config.tx_type = HWTSTAMP_TX_ON;
    config.rx_filter = HWTSTAMP_FILTER_ALL;

    ifr.ifr_data = reinterpret_cast<char*>(&config);
    // Needs CAP_NET_ADMIN. The driver may substitute a filter it
    // supports: check config.rx_filter after the call
    if (ioctl(sock_fd_, SIOCSHWTSTAMP, &ifr) < 0) {
        throw std::runtime_error("NIC does not support HW timestamping");
    }

    // Enable SO_TIMESTAMPING
    int flags = SOF_TIMESTAMPING_RX_HARDWARE |
                SOF_TIMESTAMPING_TX_HARDWARE |
                SOF_TIMESTAMPING_RAW_HARDWARE;
    setsockopt(sock_fd_, SOL_SOCKET, SO_TIMESTAMPING,
                &flags, sizeof(flags));
}

void receive_with_timestamp() {
    char buf[2048];
    char ctrl[2048];

    struct msghdr msg = {};
    struct iovec iov = {buf, sizeof(buf)};
    msg.msg_iov = &iov;
    msg.msg_iovlen = 1;
    msg.msg_control = ctrl;
    msg.msg_controllen = sizeof(ctrl);

    recvmmsg(sock_fd_, &msg, 0);

    // Extract hardware timestamp
    for (struct cmsghdr* cmsg = CMSG_FIRSTHDR(&msg);
         cmsg != NULL;
         cmsg = CMSG_NXTHDR(&msg, cmsg)) {

        if (cmsg->cmsg_level == SOL_SOCKET &&
            cmsg->cmsg_type == SO_TIMESTAMPING) {

            struct timespec* ts =
                reinterpret_cast<struct timespec*>(CMSG_DATA(cmsg));

            // ts[0] = software, ts[2] = raw hardware timestamp (NIC clock)
            uint64_t hw_timestamp_ns = ts[2].tv_sec * 1000000000ULL +
                                       ts[2].tv_nsec;

            printf("HW timestamp: %" PRIu64 " ns\n", hw_timestamp_ns);
        }
    }
}
};

```

TCP/UDP Performance Optimization

TCP Socket Tuning

```
void optimize_tcp_socket(int sock_fd) {
    // Disable Nagle's algorithm: send small messages immediately instead
    // of waiting to coalesce them. The single most important setting.
    int flag = 1;
    setsockopt(sock_fd, IPPROTO_TCP, TCP_NODELAY, &flag, sizeof(flag));

    // Bigger buffers don't make anything faster; they absorb bursts so the
    // kernel doesn't drop data when the reader falls behind. The kernel
    // caps them at net.core.rmem_max/wmem_max and doubles the value.
    int buf_size = 4 * 1024 * 1024; // 4MB
    setsockopt(sock_fd, SOL_SOCKET, SO_RCVBUF, &buf_size, sizeof(buf_size));
    setsockopt(sock_fd, SOL_SOCKET, SO_SNDBUF, &buf_size, sizeof(buf_size));

    // Mark packets for priority queuing in the NIC / qdisc
    int priority = 6;
    setsockopt(sock_fd, SOL_SOCKET, SO_PRIORITY, &priority, sizeof(priority));
}

// TCP_QUICKACK (send ACKs immediately instead of delaying them) is NOT
// permanent: the kernel can drop back to delayed ACKs, so it must be
// re-armed after every receive (one extra, cheap system call; bypass
// stacks like Onload handle ACKs themselves)
ssize_t recv_quickack(int sock_fd, void* buf, size_t len) {
    ssize_t n = recv(sock_fd, buf, len, 0);
    int one = 1;
    setsockopt(sock_fd, IPPROTO_TCP, TCP_QUICKACK, &one, sizeof(one));
    return n;
}
```

TCP_USER_TIMEOUT is sometimes listed as a latency setting; it isn't. It sets how long transmitted data may stay unacknowledged before the kernel *aborts the connection*. It's a dead-peer detector, useful for noticing a lost exchange session quickly, but set too low it kills healthy connections during a brief network hiccup.

Zero-Copy Send (MSG_ZEROCOPY): Usually Not for Trading

MSG_ZEROCOPY lets the kernel send directly from your buffer instead of copying it. The kernel documentation says it is generally only effective for writes over about 10 KB. Below that, pinning the pages and handling the completion notification cost more than the copy saved. Orders and market data messages are tens to hundreds of bytes, so for them a normal `send` is faster.

If you do use it for large transfers: - Enable SO_ZEROCOPY once at socket setup, not per send - Don't modify or reuse the buffer until the completion notification arrives on the socket's error queue (`recvmsg(..., MSG_ERRQUEUE)`)

Real zero-copy on the hot path comes from kernel bypass (above).

Multicast Optimization

```
// Receiver: join a market data multicast group on a specific interface
int setup_multicast_receiver(const char* group_ip, uint16_t port,
                             const char* interface_ip) {
```

```

int sock_fd = socket(AF_INET, SOCK_DGRAM, 0);

int reuse = 1; // several processes can receive the same feed
setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));

// Bind to the group address so only this group's traffic arrives
struct sockaddr_in addr = {};
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
inet_pton(AF_INET, group_ip, &addr.sin_addr);
bind(sock_fd, reinterpret_cast<sockaddr*>(&addr), sizeof(addr));

// Join on an explicit interface: with INADDR_ANY the kernel picks one
// from the routing table, which on a multi-NIC host is often wrong
struct ip_mreq mreq = {};
inet_pton(AF_INET, group_ip, &mreq.imr_multiaddr);
inet_pton(AF_INET, interface_ip, &mreq.imr_interface);
setsockopt(sock_fd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));

return sock_fd;
}

// Sender-side options (only matter when you publish multicast):
//   IP_MULTICAST_IF   - which interface to send from
//   IP_MULTICAST_TTL   - how many router hops the packets may cross
//   IP_MULTICAST_LOOP - 0 to stop your own packets looping back to local sockets

```

SIMD and Vectorization

Manual SIMD with Intrinsics

```

#include <immintrin.h>

// Sum array using AVX2 (process 8 floats at once)
float simd_sum(const float* data, size_t count) {
    __m256 sum_vec = _mm256_setzero_ps();

    size_t i = 0;
    // Process 8 elements at a time
    for (; i + 8 <= count; i += 8) {
        __m256 v = _mm256_loadu_ps(&data[i]);
        sum_vec = _mm256_add_ps(sum_vec, v);
    }

    // Horizontal sum
    __m128 sum_high = _mm256_extractf128_ps(sum_vec, 1);
    __m128 sum_low = _mm256_castps256_ps128(sum_vec);
    sum_low = _mm_add_ps(sum_low, sum_high);
    sum_low = _mm_hadd_ps(sum_low, sum_low);
    sum_low = _mm_hadd_ps(sum_low, sum_low);

    float result = _mm_cvtss_f32(sum_low);
}

```

```

    // Handle remaining elements
    for (; i < count; ++i) {
        result += data[i];
    }

    return result;
}

```

Two things to know about this loop: - **One accumulator is latency-bound.** Each `_mm256_add_ps` waits for the previous one (~4 cycles), while the CPU could start two per cycle. Using 4-8 independent accumulators and adding them at the end is typically several times faster on data in cache. - **The result differs slightly from a scalar loop.** Floating-point addition isn't associative, and the vector version adds in a different order. That's also why compilers won't vectorize a plain float sum by themselves unless you allow reassociation (`-ffast-math` or `-fassociative-math`).

A VWAP example, and how data layout decides whether SIMD pays off, is in interview question 4 below.

Auto-Vectorization

```

// Compiler can auto-vectorize this
void add_arrays(float* __restrict__ result,
               const float* __restrict__ a,
               const float* __restrict__ b,
               size_t count) {
    // __restrict__ promises the arrays don't overlap, so the compiler
    // doesn't need a runtime overlap check (or a scalar fallback)
    for (size_t i = 0; i < count; ++i) {
        result[i] = a[i] + b[i];
    }
}

// Vectorization is on at -O2 (GCC 12+) and -O3. Choose the instruction set
// with -march (see Chapter 0 on -march=native vs a named target).
// Check what happened:
// g++ -O3 -march=x86-64-v3 -fopt-info-vec-missed file.cpp
// clang++ -O3 -march=x86-64-v3 -Rpass-missed=loop-vectorize file.cpp
// `#pragma omp simd` asks for vectorization explicitly, but is ignored
// unless you compile with -fopenmp-simd (or -fopenmp)

```

Profiling and Measurement

CPU Performance Counters (perf)

```

# Count cache misses
perf stat -e cache-misses,cache-references ./trading_app

# Profile with call graph
perf record -g ./trading_app
perf report

# Find hotspots
perf top

# Analyze specific function

```

```
perf record -e cycles -g -- ./trading_app
perf annotate process_order
```

Intel VTune

```
# Hotspot analysis
vtune -collect hotspots ./trading_app

# Memory access analysis
vtune -collect memory-access ./trading_app

# Micro-architecture analysis
vtune -collect uarch-exploration ./trading_app
```

Manual Instrumentation

```
// Records into a histogram; printing in a destructor on the hot path
// would cost far more than the code being measured
class ScopedTimer {
    LatencyHistogram& hist_;
    uint64_t start_;

public:
    explicit ScopedTimer(LatencyHistogram& hist)
        : hist_(hist), start_(TSCClock::ticks()) {}

    ~ScopedTimer() {
        // to_ns is one multiply; tsc_clock is the process-wide TSCClock above
        hist_.record(static_cast<uint64_t>(tsc_clock.to_ns(TSCClock::ticks() - start_)));
    }
};

LatencyHistogram process_order_hist;

void process_order() {
    ScopedTimer timer(process_order_hist);
    // ... implementation ...
}
```

The timer itself costs ~15ns (two TSC reads plus a histogram increment), so it can't measure things much shorter than that. For sub-100ns sections, measure many iterations in a microbenchmark instead.

Latency Histogram

Latencies are summarized with a histogram and read as percentiles; see interview question 1 for a log-scale histogram and HdrHistogram. Bucket boundaries must match the latencies you expect: fixed 1ps buckets put everything a sub-microsecond system does into bucket 0.

Interview Questions & Answers

1. “How do you measure and optimize tail latency (p99, p99.9)?”

Answer:

Tail latency is the latency experienced by the slowest requests and is critical in low-latency systems where every request matters.

Measurement Techniques:

1. Latency Histograms:

```
class LatencyHistogram {
    // Log-scale buckets for wide range
    static constexpr size_t NUM_BUCKETS = 100;
    std::array<std::atomic<uint64_t>, NUM_BUCKETS> buckets_{};

    // Bucket i covers [2^i, 2^(i+1)) nanoseconds
    size_t get_bucket(uint64_t latency_ns) const {
        if (latency_ns == 0) return 0;
        // Count leading zeros to find log2
        return 63 - __builtin_clzll(latency_ns);
    }

public:
    void record(uint64_t latency_ns) {
        size_t bucket = std::min(get_bucket(latency_ns), NUM_BUCKETS - 1);
        buckets_[bucket].fetch_add(1, std::memory_order_relaxed);
    }

    uint64_t percentile(double p) const {
        uint64_t total = 0;
        for (const auto& b : buckets_) {
            total += b.load();
        }

        uint64_t target = total * p / 100.0;
        uint64_t cumulative = 0;

        for (size_t i = 0; i < buckets_.size(); ++i) {
            cumulative += buckets_[i].load();
            if (cumulative >= target) {
                return 1ULL << i; // Return bucket minimum
            }
        }

        return 0;
    }

    void print_stats() const {
        printf("p50:    >= %" PRIu64 " ns\n", percentile(50));
        printf("p90:    >= %" PRIu64 " ns\n", percentile(90));
        printf("p99:    >= %" PRIu64 " ns\n", percentile(99));
        printf("p99.9: >= %" PRIu64 " ns\n", percentile(99.9));
        printf("p99.99: >= %" PRIu64 " ns\n", percentile(99.99));
    }
};
```

Power-of-2 buckets are cheap (one `clz` per sample) but coarse: a p99 anywhere from 1024 to 2047ns reports as 1024. Log-linear buckets fix this by splitting each power of two into, say, 16 or 64 linear sub-buckets; that's what `HdrHistogram` does.

With several recording threads, give each thread its own histogram and merge when reporting. One shared array of atomic counters puts adjacent buckets on shared cache lines, which then bounce between cores.

2. HDR Histogram (High Dynamic Range):

```
#include <hdr/hdr_histogram.h>

class HDRLatencyTracker {
    struct hdr_histogram* histogram_;

public:
    HDRLatencyTracker() {
        // Min=1ns, Max=1hour, 3 significant digits
        hdr_init(1, 3600000000000, 3, &histogram_);
    }

    void record(uint64_t latency_ns) {
        hdr_record_value(histogram_, latency_ns);
    }

    void print_percentiles() {
        printf("p50: %" PRIu64 " ns\n", hdr_value_at_percentile(histogram_, 50.0));
        printf("p99: %" PRIu64 " ns\n", hdr_value_at_percentile(histogram_, 99.0));
        printf("p99.9: %" PRIu64 " ns\n", hdr_value_at_percentile(histogram_, 99.9));
        printf("p99.99: %" PRIu64 " ns\n", hdr_value_at_percentile(histogram_, 99.99));
    }

    ~HDRLatencyTracker() {
        hdr_close(histogram_);
    }
};
```

Optimizing Tail Latency:

1. Eliminate Allocation in Hot Path:

```
// BAD: Allocates on every call
void process_order() {
    std::vector<Trade> trades; // Heap allocation
    // ...
}

// GOOD: Pre-allocated buffer
class OrderProcessor {
    std::array<Trade, MAX_TRADES> trade_buffer_;
    size_t trade_count_ = 0;

public:
    void process_order() {
        trade_count_ = 0; // Reset
        // Use trade_buffer_[trade_count_++]
    }
};
```

2. Pin Threads and Isolate Cores:

```
# Isolate cores 2-7 from OS scheduler
```

```
# Add to kernel boot parameters:
isolcpus=2-7 nohz_full=2-7 rcu_nocbs=2-7
```

```
# Then pin critical threads to isolated cores
```

3. Avoid System Calls and Cheap-Looking Calls:

```
// Use TSC instead of clock_gettime()
uint64_t now = __rdtsc(); // ~6-8ns

// vs
struct timespec ts;
clock_gettime(CLOCK_MONOTONIC, &ts); // ~20ns via vDSO (no system call), more jitter
```

4. Handle Interrupts:

```
# Stop irqbalance from moving interrupts back onto your cores
systemctl stop irqbalance
```

```
# Pin network card interrupts to a housekeeping core
echo 1 > /proc/irq/123/smp_affinity # bitmask 0x1 = core 0
```

```
# Disable interrupt coalescing: deliver each packet immediately instead of
# batching packets to save interrupts (adaptive coalescing adds variable delay)
ethtool -C eth0 adaptive-rx off rx-usecs 0
```

With kernel bypass and busy-polling, the NIC's receive interrupts can often be avoided altogether.

5. Monitor and Alert on Tail Latency:

```
class TailLatencyMonitor {
    LatencyHistogram hist_;
    std::atomic<uint64_t> p99_threshold_{1000000}; // 1ms default

public:
    void record_and_check(uint64_t latency_ns) {
        hist_.record(latency_ns);

        // Alert if exceeds threshold
        if (latency_ns > p99_threshold_.load(std::memory_order_relaxed)) {
            alert_tail_latency_spike(latency_ns);
        }
    }

    void update_threshold() {
        // Recalculate threshold from recent data
        uint64_t new_threshold = hist_.percentile(99.0) * 1.5;
        p99_threshold_.store(new_threshold, std::memory_order_relaxed);
    }
};
```

Key Interview Points: - Measure p99, p99.9, p99.99, not just average - Use histograms for accurate percentile calculation - Beware coordinated omission: a load generator that waits for each response before sending the next under-samples exactly the slow periods; send on a fixed schedule and measure from the intended send time - Tail latency often caused by: allocation, page faults, interrupts, context switches, CPU frequency and idle-state (C-state) transitions, cold caches after idle periods - Eliminate non-deterministic operations from hot path - Isolate critical threads from OS interference

2. “Explain kernel bypass and when you would use it.”

Answer:

Kernel bypass allows user-space applications to access network hardware directly, eliminating kernel overhead. The two paths are compared in Kernel Bypass & User-Space Networking above: roughly 5-20 s per packet through the kernel versus 0.5-2 s with bypass.

Technologies:

1. DPDK (Data Plane Development Kit):

```
// Polls NIC directly from user space
uint16_t nb_rx = rte_eth_rx_burst(port_id, queue_id, pkts, 32);
// No system calls, no interrupts, no kernel
```

Advantages: - Roughly 5-10x lower latency than the kernel stack, and far less jitter - Zero-copy operation
- Deterministic performance - Full control over packet processing

Disadvantages: - Dedicated CPU cores (100% busy polling) - No OS network stack (implement your own TCP/IP, or use a user-space stack) - Requires supported NICs and drivers - Complex to set up and debug; standard tools like tcpdump don't see the traffic

2. OpenOnload (AMD, formerly Solarflare):

```
# Transparent acceleration via LD_PRELOAD
onload --profile=latency ./my_app

# Application uses standard sockets, but gets kernel bypass
```

Advantages: - Minimal code changes - Standard socket API - Selective acceleration

3. Mellanox VMA (Messaging Accelerator):

```
# Similar to OpenOnload
LD_PRELOAD=libvma.so ./my_app
```

When to Use Kernel Bypass:

YES - Use when: - Ultra-low latency required (< 10 s) - High packet rate (> 1M packets/sec) - Dedicated hardware available - Can dedicate CPU cores - Examples: HFT trading, market data processing

NO - Don't use when: - General-purpose application - Latency tolerance > 100 s - Limited CPU resources
- Need standard OS networking features - Examples: Web services, databases, most applications

Implementation Considerations:

```
class KernelBypassReceiver {
    void init() {
        // Pin to a core isolated with isolcpus/nohz_full/rcu_nocbs
        // (pin_thread_to_core from Chapter 2)
        pin_thread_to_core(4);

        // No real-time priority: nothing else is scheduled on an isolated
        // core, and a SCHED_FIFO thread that never sleeps starves kernel
        // work there (see Chapter 2, "Setting Thread Priority")
    }

    void receive_loop() {
        // Busy poll forever - 100% CPU usage
    }
}
```

```

while (running_) {
    uint16_t nb_rx = rte_eth_rx_burst(port_, queue_, pkts_, 32);
    for (uint16_t i = 0; i < nb_rx; ++i) {
        process_packet(pkts_[i]);
    }
    rte_pktmbuf_free_bulk(pkts_, nb_rx);
    // No sleep, no yield - continuous polling
}
};

```

Key Interview Points: - Bypasses kernel for direct hardware access - Trades CPU for latency (busy polling) - Essential for HFT, less critical for most apps - Requires dedicated hardware and cores - Roughly 5-10x lower latency, and much lower jitter

3. “How do you handle packet loss and retransmission in a low-latency system?”

Answer:

In low-latency systems, we need to balance reliability with speed. Different approaches for different protocols:

UDP - Application-Level Reliability:

1. Sequence Numbers + Negative Acknowledgment:

```

struct UDPMessage {
    uint64_t sequence_num;
    uint8_t message_type;
    // ... payload ...
};

class UDPReceiver {
    uint64_t expected_seq_ = 0; // next sequence number to process
    uint64_t next_to_nack_ = 0; // lowest sequence number not yet requested
    // Out-of-order messages waiting for the gap to fill. std::map keeps the
    // example short; production code uses a pre-allocated ring indexed by
    // sequence % capacity, so the hot path never allocates
    std::map<uint64_t, UDPMessage> pending_;

public:
    void on_receive(const UDPMessage& msg) {
        if (msg.sequence_num < expected_seq_) {
            return; // Duplicate (e.g. a retransmission we no longer need)
        }

        if (msg.sequence_num > expected_seq_) {
            // Gap detected - store for later
            pending_.emplace(msg.sequence_num, msg);

            // NACK only the part of the gap not already requested; without
            // this, every packet after a gap re-requests the whole gap
            uint64_t from = std::max(expected_seq_, next_to_nack_);
            uint64_t to = msg.sequence_num - 1;
            if (from <= to) {
                send_nack(from, to);
            }
        }
    }
};

```

```

    }
    // Everything up to this message is now either requested or here
    next_to_nack_ = std::max(next_to_nack_, msg.sequence_num + 1);
    // (A timer should re-NACK if the retransmission is lost too)
    return;
}

// In order - process immediately, then drain anything now in sequence
process_message(msg);
++expected_seq_;
for (auto it = pending_.begin();
     it != pending_.end() && it->first == expected_seq_;
     it = pending_.erase(it)) {
    process_message(it->second);
    ++expected_seq_;
}
}

void send_nack(uint64_t start, uint64_t end) {
    NACKMessage nack{start, end};
    send_to_server(nack); // Request retransmission
}
};

```

2. Forward Error Correction (FEC):

```

// Send redundant packets to recover from loss without retransmission
class FECEncoder {
public:
    // Simplest scheme: one XOR parity packet per group of packets.
    // The receiver can rebuild any ONE lost packet per group by XORing
    // the parity with the packets that did arrive.
    void send_group_with_parity(const std::array<Packet, 4>& group) {
        Packet parity{};
        for (const auto& pkt : group) {
            send_packet(pkt);
            xor_into(parity, pkt);
        }
        send_packet(parity); // 25% overhead, survives 1 loss in 4
    }
};

// Recovering up to K losses from N data + K parity packets needs a real
// erasure code such as Reed-Solomon, which costs more CPU per packet

```

3. Heartbeats + Gap Detection:

```

class HeartbeatMonitor {
    std::chrono::steady_clock::time_point last_heartbeat_;

public:
    void on_heartbeat() {
        last_heartbeat_ = std::chrono::steady_clock::now();
    }

    void check_liveness() {
        auto now = std::chrono::steady_clock::now();
    }
};

```

```

    auto elapsed = now - last_heartbeat_;

    if (elapsed > std::chrono::milliseconds(100)) {
        // Connection likely dead - reconnect or request snapshot
        request_snapshot();
    }
}
};

```

TCP - Tuning for Low Latency:

Order entry to an exchange runs over a long-lived TCP session. The settings that matter are covered in TCP Socket Tuning above: - `TCP_NODELAY`: send immediately (disable Nagle) - `TCP_QUICKACK`: re-armed after every receive, since it isn't permanent

Settings that look relevant but aren't: - **TCP Fast Open** saves a round trip on *connection setup*, which a session opened once per day doesn't need. (It's also enabled differently: `TCP_FASTOPEN` on the listening socket, `TCP_FASTOPEN_CONNECT` on a client.) - **Congestion control** (cubic, BBR) governs how fast bulk data ramps up. Small messages on an uncongested colocation link rarely hit its limits. - `TCP_USER_TIMEOUT` controls when a stuck connection is *killed*, not how fast data is retransmitted.

The retransmission timeout itself has a floor of 200ms on Linux (`TCP_RTO_MIN`), which is why a lost packet on a TCP order session is so expensive, and why bypass stacks and exchange-side recovery matter.

Multicast with Reliability (PGM - Pragmatic General Multicast):

PGM (RFC 3208) is a standard reliable-multicast protocol: receivers NACK missing packets and the sender retransmits them to the whole group. Exchange market data feeds generally don't use it; they rely on the redundant A/B lines, retransmission servers and snapshots described below.

Recovery Strategies:

1. Snapshot + Incremental:

```

class MarketDataRecovery {
    OrderBook book_;
    std::vector<Update> buffered_; // reserve()d at startup
    bool recovering_ = false;

public:
    void on_gap_or_connection_loss() {
        recovering_ = true;
        buffered_.clear();
        // 1. Keep receiving incrementals and BUFFER them (next function)
        // 2. Then request the snapshot. Order matters: if you snapshot
        //    first and subscribe after, updates in between are lost.
        request_snapshot_async();
    }

    void on_incremental(const Update& update) {
        if (recovering_) {
            buffered_.push_back(update);
        } else {
            apply_update(update);
        }
    }

    void on_snapshot(const OrderBook& snapshot) {

```

```

    book_ = snapshot; // state as of sequence number snapshot.last_seq
    // 3. Replay buffered updates the snapshot doesn't already include
    for (const auto& update : buffered_) {
        if (update.seq > snapshot.last_seq) {
            apply_update(update);
        }
    }
    // (If the buffer's first seq is > snapshot.last_seq + 1, there is
    // still a gap: request a newer snapshot)
    recovering_ = false;
}
};

```

2. A/B Line Arbitration:

Exchanges publish the same feed, with the same sequence numbers, on two independent multicast lines (“A” and “B”). The receiver takes each message from whichever line delivers it first:

```

class FeedArbitrator {
    uint64_t next_seq_ = 1;

public:
    // Called for every packet from line A or line B
    void on_packet(const Message& msg) {
        if (msg.seq < next_seq_) {
            return; // Already processed (the other line was faster)
        }
        if (msg.seq > next_seq_) {
            // Missing on the faster line so far: hold it briefly, the other
            // line may still deliver next_seq_. If neither does, recover.
            buffer_or_start_recovery(msg);
            return;
        }
        process_message(msg);
        ++next_seq_;
        drain_buffered_in_sequence();
    }
};

```

A message is only lost if *both* lines drop it, and taking the faster copy also shaves latency.

Key Trade-offs:

Approach	Latency	Reliability	Complexity
A/B line arbitration	Lowest (fastest copy wins)	High (fails only if both lines lose it)	Low
UDP + NACK/retransmission	Low normally, a round trip when a packet is lost	High	Medium
UDP + FEC	Low (no round trip)	Medium (limited losses per group)	High
Snapshot recovery	Slow (last resort)	Complete resync	Medium
TCP	Medium (200ms stall on loss)	High	Low

Key Interview Points: - UDP gives control but requires application-level reliability - Use sequence numbers to detect gaps - NACK-based recovery vs FEC - TCP can be tuned for lower latency - Always have recovery mechanism (snapshots) - Monitor packet loss metrics

4. “Explain how SIMD can improve performance and give an example.”

Answer:

SIMD (Single Instruction Multiple Data) processes multiple data elements with one instruction, exploiting data parallelism.

Concept:

Scalar: one add instruction per element

```
C[0] = A[0] + B[0]
C[1] = A[1] + B[1]
C[2] = A[2] + B[2]
C[3] = A[3] + B[3]
4 instructions (and 4 loads, 4 stores)
```

SIMD (SSE, 4 floats per register):

```
[C0 C1 C2 C3] = [A0 A1 A2 A3] + [B0 B1 B2 B3]
1 instruction, same latency as one scalar add
```

A CPU can issue about as many vector instructions per cycle as scalar ones, so doing 4, 8 or 16 elements per instruction gives up to that much speed-up, as long as memory bandwidth keeps up.

SIMD Instruction Sets: - SSE (128-bit): 4 floats, 2 doubles - AVX2 (256-bit): 8 floats, 4 doubles - AVX-512 (512-bit): 16 floats, 8 doubles

Example 1: Array Summation

See `simd_sum` in SIMD and Vectorization above, including why it needs several accumulators to reach full speed.

Example 2: VWAP Calculation (Real Trading Use Case)

Data layout decides whether SIMD pays off. With an array of `{price, volume}` structs, filling a vector register means gathering fields from 8 different structs, which can cost more than the arithmetic saves. Store each field in its own array (structure of arrays, Chapter 1) and the loads become single instructions.

Prices are integers here, in ticks (the instrument’s minimum price increment), as they usually are in trading systems: floats can’t represent most decimal prices exactly, and a float sum of millions of `price * volume` products loses digits.

```
#include <immintrin.h>
#include <cstdint>

// Volume-Weighted Average Price over trades stored as structure of arrays
struct Trades {
    const int32_t* price_ticks; // price / tick_size
    const int32_t* volume;
    size_t count;
};

// Scalar version: exact integer arithmetic
double vwap_scalar(const Trades& t, double tick_size) {
    int64_t pv_sum = 0; // sum of price * volume, in ticks * shares
```

```

    int64_t vol_sum = 0;
    for (size_t i = 0; i < t.count; ++i) {
        pv_sum += int64_t{t.price_ticks[i]} * t.volume[i];
        vol_sum += t.volume[i];
    }
    return static_cast<double>(pv_sum) / vol_sum * tick_size;
}

// AVX2 version: 4 trades per iteration with 64-bit accumulators
// (32-bit lanes would overflow: price * volume easily exceeds 2^31)
double vwap_avx2(const Trades& t, double tick_size) {
    __m256i pv_acc = _mm256_setzero_si256();
    __m256i vol_acc = _mm256_setzero_si256();

    size_t i = 0;
    for (; i + 4 <= t.count; i += 4) {
        // Load 4 prices and 4 volumes, widen 32-bit -> 64-bit lanes
        __m256i p = _mm256_cvtepi32_epi64(_mm_loadu_si128(
            reinterpret_cast<const __m128i*>(t.price_ticks + i)));
        __m256i v = _mm256_cvtepi32_epi64(_mm_loadu_si128(
            reinterpret_cast<const __m128i*>(t.volume + i)));

        // _mm256_mul_epi32 multiplies the low 32 bits of each 64-bit lane
        // (signed) into a full 64-bit product
        pv_acc = _mm256_add_epi64(pv_acc, _mm256_mul_epi32(p, v));
        vol_acc = _mm256_add_epi64(vol_acc, v);
    }

    // Reduce the 4 lanes to scalars
    alignas(32) int64_t pv[4], vol[4];
    _mm256_store_si256(reinterpret_cast<__m256i*>(pv), pv_acc);
    _mm256_store_si256(reinterpret_cast<__m256i*>(vol), vol_acc);
    int64_t pv_sum = pv[0] + pv[1] + pv[2] + pv[3];
    int64_t vol_sum = vol[0] + vol[1] + vol[2] + vol[3];

    // Handle remaining elements
    for (; i < t.count; ++i) {
        pv_sum += int64_t{t.price_ticks[i]} * t.volume[i];
        vol_sum += t.volume[i];
    }

    // Same result as the scalar version, exactly (integer arithmetic)
    return static_cast<double>(pv_sum) / vol_sum * tick_size;
}

```

For a loop this simple, `-O3 -march=x86-64-v3` usually vectorizes the scalar version by itself. Check the compiler output before writing intrinsics.

Example 3: Order Book (Finding a Price Level)

```

// Find the first level whose price is >= target in an ascending array
// of price levels (prices in integer ticks). Returns n if none.
size_t find_price_level_simd(const int32_t* prices, size_t n, int32_t target) {
    // Compare against target - 1 so "greater than" means ">= target"
    // (AVX2 has signed > and == comparisons, but no >=)

```

```

__m256i threshold = _mm256_set1_epi32(target - 1);

size_t i = 0;
for (; i + 8 <= n; i += 8) {
    __m256i p = _mm256_loadu_si256(
        reinterpret_cast<const __m256i*>(prices + i));

    // Each lane becomes all-ones where prices[i] > target - 1
    __m256i gt = _mm256_cmpgt_epi32(p, threshold);

    // One bit per 32-bit lane
    int mask = _mm256_movemask_ps(_mm256_castsi256_ps(gt));

    if (mask != 0) {
        // First matching lane = number of trailing zero bits
        return i + __builtin_ctz(mask);
    }
}

// Scalar tail
for (; i < n; ++i) {
    if (prices[i] >= target) return i;
}
return n;
}

```

Order books keep a few dozen active levels in a small contiguous array, so a scan like this touches a couple of cache lines, which is often faster than a tree or binary search.

When SIMD Helps:

Good use cases: - Processing contiguous arrays of numbers (prices, volumes) - Searching small arrays (order book levels, tables of IDs) - Parsing and validating text (e.g. simdjson, FIX tag scanning) - Checksums and hashing

Poor use cases: - Irregular, pointer-chasing data access - Heavy branching logic per element - Data stored as array-of-structs with only one field needed - Loop-carried dependencies (each step needs the previous result)

Key Interview Points: - SIMD processes multiple data elements per instruction - AVX2 does 8 float or 4 double operations per instruction; AVX-512 doubles that - Data layout (structure of arrays) matters more than intrinsics - Unaligned loads (`loadu`) are fine on modern CPUs; only loads that straddle a cache line cost extra - On some Intel server CPUs (Skylake-SP era), heavy AVX-512 use lowers the core's clock frequency for a while, which can slow the scalar code that follows; measure before enabling it on a latency-critical core - Floating-point SIMD reorders additions, so results differ in the last bits from scalar code - Modern compilers auto-vectorize simple loops; check with `-fopt-info-vec` before writing intrinsics

5. "How would you debug a latency spike in production?"

Answer:

A systematic approach to identifying and resolving latency spikes:

Step 1: Gather Data

```

// Instrument critical paths: a few TSC reads into a fixed array, so the
// instrumentation itself doesn't allocate or calibrate on the hot path

```

```

class LatencyInstrument {
    static constexpr size_t MAX_CHECKPOINTS = 8;

    struct Checkpoint { const char* name; uint64_t ticks; };
    std::array<Checkpoint, MAX_CHECKPOINTS> checkpoints_;
    size_t count_ = 0;
    uint64_t start_ = TSCClock::ticks();

public:
    void checkpoint(const char* name) {
        if (count_ < MAX_CHECKPOINTS) {
            checkpoints_[count_++] = {name, TSCClock::ticks() - start_};
        }
    }

    ~LatencyInstrument() {
        uint64_t total = TSCClock::ticks() - start_;

        if (total > LATENCY_THRESHOLD_TICKS) [[unlikely]] {
            // Hand the record to a background logger thread (e.g. over an
            // SPSC queue); never format and write from the hot thread
            slow_path_log(checkpoints_.data(), count_, total);
        }
    }
};

void process_order() {
    LatencyInstrument lat;

    parse_message();
    lat.checkpoint("parse");

    validate_order();
    lat.checkpoint("validate");

    send_to_exchange();
    lat.checkpoint("send");
}

```

Step 2: Check System Metrics

```

# CPU usage and context switches
sar -u 1 10
sar -w 1 10

# Packet drops and errors on the NIC and in the kernel
ethtool -S eth0 | grep -iE 'drop|miss|err'
netstat -su    # UDP receive buffer errors

# Check for CPU throttling (frequency, thermal limits)
cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_cur_freq
turboostat --quiet sleep 5    # frequency, C-state residency, SMI count per core

# IRQ distribution
cat /proc/interrupts

```

```
# Check NUMA balancing
cat /proc/sys/kernel/numa_balancing
```

Step 3: Profile the Application

```
# CPU profiling
perf record -g -p <PID> -- sleep 10
perf report

# Check for cache misses
perf stat -e cache-misses,cache-references -p <PID> -- sleep 10

# Memory bandwidth (uncore events are system-wide: -a, not -p)
perf stat -a -e uncore_imc/data_reads/,uncore_imc/data_writes/ -- sleep 10

# Which kernel events hit the latency-critical core (core 4 here)
perf record -C 4 -e 'irq:*,sched:sched_switch' -- sleep 10

# Context switches
perf stat -e context-switches -p <PID> -- sleep 10
```

Step 4: Common Causes

1. Memory Allocation:

```
// Check for allocations in hot path
void process_tick_bad() {
    std::vector<Trade> trades; // Allocation!
    // ...
}

void process_tick_good() {
    thread_local std::vector<Trade> trades;
    trades.clear(); // Reuse allocation
    // ...
}
```

2. Lock Contention:

```
# Use perf to find lock contention
perf record -e syscalls:sys_enter_futex -ag -- sleep 10
perf report

# Or use flamegraphs
perf script | stackcollapse-perf.pl | flamegraph.pl > flame.svg
```

3. Page Faults:

```
# Check for page faults
perf stat -e page-faults,minor-faults,major-faults -p <PID> -- sleep 10

// At startup, before trading begins:

// Lock all current and future memory: no paging out, and future
// mappings are faulted in when created
mlockall(MCL_CURRENT | MCL_FUTURE);

// Pre-fault buffers allocated before mlockall, or use MAP_POPULATE
```

```
for (size_t i = 0; i < size; i += 4096) {
    buffer[i] = 0;    // Touch each page
}
```

A first touch of a new page is a minor fault (~1µs, plus zeroing the page). Transparent huge pages can add larger, rarer stalls when the kernel compacts memory to create them (`khugepaged`); many low-latency setups set THP to `madvise` or `never` and use explicitly reserved huge pages (`hugetlbfs`) for large buffers, which also cuts TLB misses.

4. CPU Frequency and Idle States:

```
# Check current frequency
cpupower frequency-info

# Set to performance mode
cpupower frequency-set -g performance

# Disable turbo boost (for consistency)
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo

# Deep idle states (C-states) take tens of µs to wake from.
# Limit them in BIOS or on the kernel command line:
#   intel_idle.max_cstate=1 processor.max_cstate=1
# (a busy-polling core never idles anyway; this matters for cores that block)
```

System Management Interrupts (SMIs) are handled by firmware and stall a core for tens to hundreds of µs, invisibly to the OS. `turbostat` shows an SMI count; BIOS “low latency” profiles reduce them.

5. IRQ Affinity:

```
# Check which CPUs handle network interrupts
cat /proc/interrupts | grep eth0

# Move network IRQs off the latency-critical cores (see question 1)
echo 1 > /proc/irq/123/smp_affinity # bitmask 0x1 = core 0 only
```

6. NUMA Node Mismatch:

```
# Check NUMA topology, and which node the NIC is attached to
numactl --hardware
cat /sys/class/net/eth0/device/numa_node

# Bind to specific NUMA node
numactl --cpunodebind=0 --membind=0 ./trading_app

# Stop the kernel migrating pages between nodes behind your back
echo 0 > /proc/sys/kernel/numa_balancing

// Or in code (libnuma)
numa_run_on_node(0);
numa_set_preferred(0);
```

Step 5: Reproduce and Fix

Replay recorded production traffic (e.g. a packet capture of the market data feed) at its original rate, or faster, on a test system configured like production, and compare the latency histograms before and after a fix. Load tests should send on a fixed schedule rather than waiting for each response, or they hide the slow periods (coordinated omission, see question 1).

```

// Replay a captured feed through the real processing path
void replay_test(const std::vector<CapturedPacket>& capture, LatencyHistogram& hist) {
    const uint64_t t0 = TSCClock::ticks();
    for (const auto& pkt : capture) {
        // Wait until this packet's original arrival time
        const uint64_t due = t0 + pkt.offset_ticks;
        while (TSCClock::ticks() < due) cpu_relax();

        on_packet(pkt);

        // Measure from when the packet was due, not when we got to it,
        // so time spent behind schedule is counted
        hist.record(static_cast<uint64_t>(tsc_clock.to_ns(TSCClock::ticks() - due)));
    }
}

```

Monitoring Dashboard:

Use the `TailLatencyMonitor` from question 1 in production: record every latency into a histogram on the hot thread, and have a separate thread periodically read, publish and reset it.

Key Interview Points: - Instrument code to isolate slow sections - Use system tools (perf, turbostat, ethtool) to check the machine, not just the code - Common causes: allocation, locks, page faults, CPU frequency/C-states, interrupts, SMIs - Reproduce under stress to expose issues - Monitor percentiles, not just averages - Have both logging and metrics

Summary

Low-latency techniques require understanding the full stack:

- **Hardware:** CPU cache, NUMA, IRQs
- **Kernel:** System calls, networking, scheduling
- **Application:** Memory management, lock-free algorithms
- **Measurement:** Profiling, instrumentation, percentiles

The key is measuring first, optimizing based on data, and understanding trade-offs between latency, throughput, and CPU usage.

Chapter 5 applies these techniques to an order book and a market data parser, and covers cache warming and microbenchmarking pitfalls.

Contents

Chapter 5: Trading System Components & Measurement	1
Table of Contents	1
Order Book Design	1
Operations and Their Frequency	2
Choosing Data Structures	2
Implementation: Level-3 Messages to Level-2 Book	2
Binary Exchange Protocols	5
Why Binary	5
Framing: MoldUDP64	5
Parsing Fixed-Offset Fields	5
SBE	7
When You Must Parse FIX	7
Cache Warming	7
The Problem	7
Techniques	8
Microbenchmarking Pitfalls	8
1. The Compiler Deletes the Work	9
2. Throughput Is Not Latency	9
3. Everything Is Hot	9
4. Timer Overhead and Resolution	9
5. Noisy Machines	10
6. Changes You Didn't Make	10
7. Benchmarking the Wrong Thing	10
Interview Questions & Answers	10
1. "Design an order book for a market-by-order feed."	10
2. "How would you parse a market data feed as fast as possible?"	10
3. "Our strategy is fast in tests but the first order after a quiet period is slow. Why, and what would you do?"	11
4. "A microbenchmark says this function takes 2ns, but in production the same code takes 200ns. What's going on?"	11

Chapter 5: Trading System Components & Measurement

The earlier chapters cover general techniques. This chapter applies them to the pieces every trading system has (an order book and a market data parser), and to two practical problems that decide whether those pieces are fast in production: keeping a rarely-used path warm, and measuring performance without fooling yourself.

Table of Contents

1. Order Book Design
 2. Binary Exchange Protocols
 3. Cache Warming
 4. Microbenchmarking Pitfalls
 5. Interview Questions & Answers
-

Order Book Design

An order book holds the resting buy orders (bids) and sell orders (asks) for one instrument. Market data feeds describe it in one of two forms:

- **Level 2 (market by price):** total quantity at each price level. Updates look like “bid 100.25 now has 1,500”.
- **Level 3 (market by order):** every individual order. Updates look like “order 12345 added: buy 200 @ 100.25”, “order 12345 executed 50”, “order 12345 cancelled”. You build the per-level totals yourself.

This section builds a book from a level-3 feed such as Nasdaq ITCH, since that’s the harder case and the usual interview question.

Operations and Their Frequency

A level-3 book must support:

Operation	Input	Frequency
Add order	id, side, price, quantity	Very high
Cancel / partial cancel	id, quantity	Very high (most orders are cancelled, not filled)
Execute	id, quantity	Lower
Delete	id	High
Replace	old id, new id, price, quantity	Medium
Best bid / ask	-	After every update

Two lookups dominate: **order id** → **order** (every message except add starts with it) and **price** → **level** (to update the level’s total). Almost all activity happens within a few ticks of the best prices.

Choosing Data Structures

Price levels:

Structure	Update	Best price	Notes
<code>std::map<Price, Level></code>	$O(\log n)$, pointer chasing, allocates per level	$O(1)$ (begin)	The textbook answer; slow in practice
Sorted <code>std::vector</code> of levels, best at the back	$O(\text{distance from best})$, memmove	$O(1)$	Good: activity is near the best, so shifts are short
Array indexed by price (in ticks)	$O(1)$	$O(1)$ usually; scan when the best level empties	Fastest, if prices stay within a known band

Prices must be integers (ticks), which makes direct indexing possible. A daily price band of a few thousand ticks is only a few tens of KB per side.

Order lookup: a hash map from order id to order. `std::unordered_map` allocates a node per insert, so production code uses an open-addressing table (e.g. `absl::flat_hash_map` or a custom one) sized at startup. Some feeds make this even simpler: ITCH order reference numbers increase through the day, so an array indexed by `id - first_id_of_the_day` can replace the hash map entirely.

Implementation: Level-3 Messages to Level-2 Book

```
#include <algorithm>
#include <cstdint>
#include <unordered_map>
```

```

#include <vector>

enum class Side : uint8_t { Buy, Sell };

// Aggregates order-by-order messages into per-level quantities.
// Prices are integer ticks within [min_price, min_price + num_levels).
class OrderBook {
public:
    OrderBook(int32_t min_price, int32_t num_levels, size_t max_orders)
        : min_price_(min_price),
          max_price_(min_price + num_levels - 1),
          bid_qty_(num_levels, 0),
          ask_qty_(num_levels, 0),
          best_bid_(min_price - 1),           // sentinel: no bids
          best_ask_(max_price_ + 1) {        // sentinel: no asks
        orders_.reserve(max_orders);         // no rehashing during the day
    }

    bool add(uint64_t id, Side side, int32_t price, uint32_t qty) {
        if (price < min_price_ || price > max_price_) {
            return false; // outside the band: needs a fallback structure
        }
        orders_.emplace(id, Order{price, qty, side});
        level(side, price) += qty;
        if (side == Side::Buy) {
            best_bid_ = std::max(best_bid_, price);
        } else {
            best_ask_ = std::min(best_ask_, price);
        }
        return true;
    }

    // Partial or full execution/cancel: remove `qty` from the order
    void reduce(uint64_t id, uint32_t qty) {
        auto it = orders_.find(id);
        if (it == orders_.end()) return;
        Order& o = it->second;
        qty = std::min(qty, o.qty);
        o.qty -= qty;
        remove_qty(o.side, o.price, qty);
        if (o.qty == 0) orders_.erase(it);
    }

    void remove(uint64_t id) {
        auto it = orders_.find(id);
        if (it == orders_.end()) return;
        remove_qty(it->second.side, it->second.price, it->second.qty);
        orders_.erase(it);
    }

    // Replace = delete + add under a new id (the order loses queue priority)
    void replace(uint64_t old_id, uint64_t new_id, int32_t price, uint32_t qty) {
        auto it = orders_.find(old_id);
        if (it == orders_.end()) return;

```

```

        const Side side = it->second.side;
        remove(old_id);
        add(new_id, side, price, qty);
    }

    bool has_bid() const { return best_bid_ >= min_price_; }
    bool has_ask() const { return best_ask_ <= max_price_; }
    int32_t best_bid() const { return best_bid_; }
    int32_t best_ask() const { return best_ask_; }
    uint64_t bid_qty_at(int32_t price) const { return bid_qty_[price - min_price_]; }
    uint64_t ask_qty_at(int32_t price) const { return ask_qty_[price - min_price_]; }

private:
    struct Order {
        int32_t price;
        uint32_t qty;
        Side side;
    };

    uint64_t& level(Side side, int32_t price) {
        auto& levels = (side == Side::Buy) ? bid_qty_ : ask_qty_;
        return levels[price - min_price_];
    }

    void remove_qty(Side side, int32_t price, uint32_t qty) {
        uint64_t& total = level(side, price);
        total -= qty;
        if (total != 0) return;

        // A level emptied. If it was the best, walk to the next non-empty
        // level. The walk is usually a few steps because levels near the
        // best are dense; it is bounded by the band size.
        if (side == Side::Buy && price == best_bid_) {
            while (best_bid_ >= min_price_ && bid_qty_[best_bid_ - min_price_] == 0) {
                --best_bid_;
            }
        } else if (side == Side::Sell && price == best_ask_) {
            while (best_ask_ <= max_price_ && ask_qty_[best_ask_ - min_price_] == 0) {
                ++best_ask_;
            }
        }
    }

    int32_t min_price_;
    int32_t max_price_;
    std::vector<uint64_t> bid_qty_;    // indexed by price - min_price_
    std::vector<uint64_t> ask_qty_;
    int32_t best_bid_;
    int32_t best_ask_;
    std::unordered_map<uint64_t, Order> orders_;    // see note on hash maps above
};

```

Design points to mention: - Prices outside the band (rare, but they happen: fat-finger orders, auctions) need a fallback, e.g. a small sorted vector for outliers, or re-centring the array. - Queue position:

if you also track the order of orders *within* a level (a FIFO per level, usually an intrusive doubly linked list through the order records), you can estimate where your own order sits in the queue, which matters for market making. - **One book, one thread:** the book is updated by a single thread, so it needs no locks. Other threads get snapshots, for example through a seqlock (Chapter 2). - **Memory:** everything is allocated at startup. With `std::unordered_map`, `reserve` avoids rehashing but inserts still allocate nodes, which is why production code uses an open-addressing table or an id-indexed array. - **Validation:** feeds occasionally reference unknown order ids (after a gap, or at startup). Ignore them and rely on the sequence-number gap detection (Chapter 4) to trigger recovery.

Binary Exchange Protocols

Why Binary

FIX, the traditional text protocol, encodes fields as `tag=value` separated by SOH bytes (shown here as |):

```
8=FIX.4.4|35=D|49=CLIENT|56=EXCH|11=ORD123|55=AAPL|54=1|38=100|44=150.25|...
```

Parsing it means scanning for delimiters and converting decimal text to numbers, for every field. Binary protocols put each field at a fixed offset in a fixed format, so reading a field is one load (plus a byte swap). Exchanges therefore use binary protocols for their fast paths:

Protocol	Used for	Encoding
Nasdaq TotalView-ITCH 5.0	Market data (level 3)	Fixed-length messages, big-endian integers
Nasdaq OUCH	Order entry	Fixed-length messages, big-endian
SBE (Simple Binary Encoding)	CME MDP 3.0 market data, iLink 3 order entry, others	Fixed-offset fields described by an XML schema, little-endian
FIX	Order entry at many venues, drop copies	Text <code>tag=value</code>

Framing: MoldUDP64

ITCH market data is sent over UDP multicast inside MoldUDP64 packets. Each packet carries a header and a sequence of length-prefixed messages:

MoldUDP64 header (20 bytes):

```
Session      10 bytes  ASCII
Sequence Number  8 bytes  big-endian: sequence number of the first message
Message Count  2 bytes  big-endian (0 = heartbeat, 0xFFFF = end of session)
```

Then, Message Count times:

```
Message Length  2 bytes  big-endian
Message Data    Message Length bytes (first byte = message type)
```

The sequence number is what gap detection and A/B arbitration work on (Chapter 4).

Parsing Fixed-Offset Fields

Read fields with `memcpy` into a local variable, then convert byte order. Don't cast the buffer to a struct pointer: that's UB (no such object exists there) and the fields are unaligned (Chapter 0). Compilers turn a fixed-size `memcpy` plus byte swap into a single `movbe` or `mov + bswap`.

```
#include <bit>
#include <stdint>
#include <cstring>
```

```

// Big-endian field readers (x86 is little-endian, so swap)
inline uint16_t load_be16(const char* p) {
    uint16_t v; std::memcpy(&v, p, 2); return __builtin_bswap16(v);
}
inline uint32_t load_be32(const char* p) {
    uint32_t v; std::memcpy(&v, p, 4); return __builtin_bswap32(v);
}
inline uint64_t load_be64(const char* p) {
    uint64_t v; std::memcpy(&v, p, 8); return __builtin_bswap64(v);
}
// ITCH timestamps are 6-byte big-endian integers (ns since midnight)
inline uint64_t load_be48(const char* p) {
    return (uint64_t){load_be16(p)} << 32 | load_be32(p + 2);
}
// C++23: std::byteswap does the same as the builtins

// ITCH 5.0 "Add Order - No MPID Attribution", message type 'A', 36 bytes
struct AddOrder {
    uint16_t stock_locate; // instrument index for the day
    uint64_t timestamp_ns;
    uint64_t order_ref;
    char side; // 'B' or 'S'
    uint32_t shares;
    char stock[8]; // space-padded symbol
    uint32_t price; // 4 implied decimal places: 1502500 = 150.25
};

inline AddOrder parse_add_order(const char* m) {
    AddOrder a;
    a.stock_locate = load_be16(m + 1);
    // offset 3: 2-byte tracking number, unused here
    a.timestamp_ns = load_be48(m + 5);
    a.order_ref = load_be64(m + 11);
    a.side = m[19];
    a.shares = load_be32(m + 20);
    std::memcpy(a.stock, m + 24, 8);
    a.price = load_be32(m + 32);
    return a;
}

// Walk one MoldUDP64 packet and dispatch each message by type
template<typename Handler>
bool process_packet(const char* pkt, size_t len, Handler& handler) {
    if (len < 20) return false;
    uint64_t seq = load_be64(pkt + 10);
    uint16_t count = load_be16(pkt + 18);
    if (count == 0xFFFF) { handler.on_end_of_session(); return true; }

    size_t off = 20;
    for (uint16_t i = 0; i < count; ++i, ++seq) {
        if (off + 2 > len) return false; // truncated packet
        const uint16_t msg_len = load_be16(pkt + off);
        off += 2;
        if (msg_len == 0 || off + msg_len > len) return false;
    }
}

```

```

const char* msg = pkt + off;
off += msg_len;

switch (msg[0]) {
    case 'A':
        if (msg_len >= 36) handler.on_add(seq, parse_add_order(msg));
        break;
    // 'F' add with attribution, 'E'/'C' executed, 'X' cancel,
    // 'D' delete, 'U' replace, ... follow the same pattern
    default:
        break; // types we don't need: skip by length
}
}
return true;
}

```

Points to mention: - **Check lengths before reading.** The feed is external input; a truncated or corrupt packet must not read past the buffer. This is exactly the code to fuzz (Chapter 0). - **Decode only what you use.** Parsing into a struct is convenient; the fastest handlers read just the fields they need straight from the packet buffer, which is still in L1 after the NIC delivered it. - **Keep fixed-point prices as integers.** Divide by 10,000 only for display. - **Map symbols once.** `stock_locate` is a small integer assigned per instrument for the day; index your books by it instead of hashing symbol strings. - **Switch dispatch is fine.** A `switch` on the type byte compiles to a jump table. Order the handling so the most frequent types (adds, cancels, deletes) are cheap.

SBE

SBE messages start with a small header (block length, template id, schema id, version) followed by fields at fixed offsets, in little-endian order, so on x86 no byte swap is needed. The schema is an XML file, and a code generator produces “flyweight” codec classes: thin wrappers around a buffer pointer whose getters read the field at its offset. Nothing is copied or allocated. Repeating groups (e.g. several price-level updates in one CME MDP 3.0 message) are handled by iterating over fixed-size entries.

When You Must Parse FIX

Many venues and all drop-copy connections still use FIX. To make text parsing fast: - Parse in one pass over the buffer, dispatching on the tag number; don’t build a map of all fields. - Convert integers and prices with a hand-written loop (or `std::from_chars`), never `atoi`/`strtod` (locale checks, error handling you don’t need). - Keep prices as fixed-point integers: parse 150.25 straight into 1502500 at 4 decimal places. - For outgoing messages, pre-build the constant part of each message (header, fixed tags) once and only fill in the changing fields; the checksum (tag 10) and body length (tag 9) must be recomputed.

Cache Warming

The Problem

In many strategies the code path that actually sends an order runs rarely: perhaps once per thousands of market data updates. Between orders, market data processing fills the caches with other code and data. When the strategy finally decides to trade, the order path: - misses in the instruction cache (and the micro-op cache), - misses in the data caches (order templates, session state, risk limits), - misses in the TLB, - has branches the predictor hasn’t seen recently.

The result is that the most important event of the day is also the slowest: the first order after a quiet period can take several times longer than the same path when it’s hot.

Techniques

1. Run the order path on every event, and skip only the last step. Build, risk-check and encode an order every time, and only decide at the very end whether to hand it to the NIC:

```
void on_market_data(const Tick& tick) {
    update_book(tick);
    const Signal signal = strategy_.evaluate(book_);

    // Always prepare the order, whether or not we will trade, so the
    // code and data of the order path stay in cache
    OrderMsg& msg = gateway_.prepare(signal); // fills a pre-allocated buffer
    const bool send = signal.should_trade && risk_.check(msg);
    gateway_.submit(msg, send);
}

void Gateway::submit(OrderMsg& msg, bool send) {
    finalize(msg); // sequence number, checksum, etc.
    if (send) {
        commit(msg); // advance sequence numbers, update positions
        nic_.transmit(msg); // or: the NIC API's "warm" / dry-run mode
    }
}
```

Things that make or break this: - **Dry runs must not change state.** A dry run that advances the session sequence number or counts against a risk limit corrupts the real session. Keep state changes in the part that runs only when sending. - **The branch predictor learns the dry-run outcome.** If `send` is false 9,999 times out of 10,000, the predictor will predict “don’t send” and mispredict on the one real order. Keep the final decision late and small (one branch around the transmit), so only that branch mispredicts, not the whole path. - **Some NIC APIs support this directly:** for example, some vendor send APIs have a “warm” flag that runs the send path without transmitting, so even the driver code stays hot.

2. Periodic warm-up when the market is quiet. If no market data arrives for a while (overnight in another market, a halted instrument), run a dry-run order through the path every few hundred microseconds from the same thread.

3. Touch memory up front. Pre-fault and lock all memory at startup (`mlockall`, `MAP_POPULATE`, see Chapter 4), and use huge pages for large tables to cut TLB misses.

4. Lay out code so the hot path is compact. Mark error handling `[[unlikely]]` / `__attribute__((cold))` so it moves out of line, and use profile-guided optimization (`-fprofile-generate` then `-fprofile-use`, or BOLT on the binary) to put the code that runs together next to each other. Profile with realistic traffic, including the order path.

5. Keep other work off the core. Anything else that runs on the same core (logging, housekeeping, other strategies) evicts the order path. Dedicated cores (Chapter 2) help, but the L3 cache is shared by the whole socket, so a noisy neighbor on another core can still evict your data; Intel CAT (Cache Allocation Technology) can reserve part of the L3 for your cores.

Microbenchmarking Pitfalls

Microbenchmarks answer questions like “is parser A faster than parser B?”. They are easy to get wrong in ways that produce precise-looking, meaningless numbers.

1. The Compiler Deletes the Work

If the result of the benchmarked code isn't used, the compiler may remove the code entirely, and you measure an empty loop. If the inputs are compile-time constants, it may compute the answer at compile time.

```
#include <benchmark/benchmark.h> // Google Benchmark

static void BM_ParseAddOrder(benchmark::State& state) {
    // Realistic, varied inputs prepared outside the timed loop
    const auto msgs = make_random_add_orders(4096); // not constants
    size_t i = 0;
    for (auto _ : state) {
        AddOrder a = parse_add_order(msgs[i++ & 4095].data());
        benchmark::DoNotOptimize(a); // forces the result to be "used"
    }
}

BENCHMARK(BM_ParseAddOrder);
BENCHMARK_MAIN();
```

`benchmark::DoNotOptimize(x)` tells the compiler the value is observed; `benchmark::ClobberMemory()` tells it all memory may have been read, forcing pending stores to happen. A benchmark whose result is under a nanosecond is a red flag that the work was optimized away.

2. Throughput Is Not Latency

If each iteration is independent, the CPU overlaps many iterations in its out-of-order pipeline, and the benchmark reports throughput (time per operation when many are in flight). For a latency-critical path, what matters is how long one operation takes from start to finish. Make each iteration depend on the previous one to measure that:

```
// Throughput: iterations overlap
for (auto _ : state) {
    benchmark::DoNotOptimize(hash(next_input()));
}

// Latency: a dependency chain, each input is the previous output
uint64_t x = 1;
for (auto _ : state) {
    x = hash(x);
}
benchmark::DoNotOptimize(x);
```

The two can differ by several times for the same code.

3. Everything Is Hot

A benchmark loop runs the same code on the same data millions of times: every cache is warm, and the branch predictor has learned every branch, often including patterns in the test data. Production runs the code once, cold, between other work (see Cache Warming). Consequences: - Use varied, realistic inputs (e.g. replayed from a capture) so branches aren't perfectly predicted. - For a cold-cache number, evict the caches between iterations (touch a large buffer) or measure in the real system with TSC timestamps. - A data structure that fits in L1 in the benchmark may not fit next to everything else in production.

4. Timer Overhead and Resolution

Reading the TSC costs ~25 cycles and `clock_gettime` ~20ns, so timing a 5ns operation individually measures mostly the timer. Time a batch and divide, or let the framework do it (Google Benchmark runs enough

iterations to amortize its timer). To time single events in production, subtract the measured cost of an empty timed region.

5. Noisy Machines

CPU frequency changes, turbo, other processes, interrupts and SMT siblings all add variance: - Run on an isolated core with the **performance** governor and turbo disabled, or at least pinned (**taskset**). - Repeat the run several times (**--benchmark_repetitions=10**) and look at the spread, not one number. Report medians or percentiles. - Compare against a baseline measured in the same session.

6. Changes You Didn't Make

Code layout effects can move a benchmark by 5-20%: an unrelated change shifts a hot loop across a 64-byte boundary, or changes which branches alias in the predictor. Before believing a small improvement: - Check hardware counters (**perf stat -e cycles,instructions,branch-misses,L1-dcache-load-misses**): a real improvement usually shows as fewer instructions or misses, not just fewer cycles. - Re-run with a different build (e.g. link order changed) to see if the gain survives. - Tools like Stabilizer and **-falign-loops** experiments exist precisely because of this effect.

7. Benchmarking the Wrong Thing

A 30% faster parser matters little if parsing is 2% of the tick-to-trade latency. Measure the whole path in the real system first (Chapter 4's instrumentation and histograms), find where the time goes, and microbenchmark only that part.

Interview Questions & Answers

1. "Design an order book for a market-by-order feed."

Answer:

Start from the access pattern: - Every message except "add" arrives with only an order id, so I need **id** → **order** lookup: a pre-sized open-addressing hash map, or, if ids are dense and increasing like ITCH order references, a plain array indexed by id. - Each order points to its price level, and I need **price** → **level**: since prices are integer ticks within a daily band, an array indexed by **price - min_price** gives O(1) updates. - I track **best bid/ask** explicitly. Adds update them with a min/max; when the best level empties, I scan to the next non-empty level, which is short because activity is concentrated near the top.

Then the engineering points: - All memory allocated at startup; no allocation per message. - One thread owns the book, so no locks; other threads read snapshots through a seqlock or receive updates over an SPSC queue. - Prices outside the band go to a fallback structure. - If I need queue position (market making), each level keeps a FIFO of its orders as an intrusive doubly linked list, so removing an order from the middle is O(1). - Sequence numbers detect gaps; on a gap, the book is rebuilt from a snapshot.

Why not **std::map**? It's O(log n) with a pointer chase per level and an allocation per new level: correct, but slow and unpredictable. A sorted vector with the best level at the back is a good middle ground when the price range is unknown.

2. "How would you parse a market data feed as fast as possible?"

Answer: - **Binary protocol, fixed offsets**: read each field with **memcpy** + byte swap (one instruction after optimization). No struct-pointer casts: that's UB and misaligned. - **Zero copy**: parse directly from the receive buffer (kernel bypass delivers it straight into memory the application owns). - **Decode lazily**: dispatch on the message type byte with a **switch**, and read only the fields the strategy needs. - **Integers**

only: keep prices as fixed-point integers; map instruments by the feed's integer index (ITCH `stock_locate`), never by hashing symbol strings on the hot path. - **Validate lengths** against the packet size before reading any field, and fuzz the parser. - **Sequence numbers first:** check the packet's sequence number before processing, for gap detection and A/B arbitration. - **Measure:** benchmark with a captured feed, not synthetic constant messages, so branch prediction and caches behave realistically.

3. “Our strategy is fast in tests but the first order after a quiet period is slow. Why, and what would you do?”

Answer:

The order path is cold. While only market data was processed, the order code, its data (templates, session state, risk limits), its TLB entries and its branch history were evicted or overwritten. The first order pays for instruction cache misses, data cache misses, page walks and mispredictions.

Fixes: 1. **Exercise the order path on every event** (or periodically when quiet): build, risk-check and encode an order each time and skip only the final transmit, using the NIC API's warm/dry-run mode if it has one. Dry runs must not change state (sequence numbers, risk counters). 2. Keep the final send/don't-send decision as one late branch so the predictor's bias costs one misprediction, not many. 3. Compact the hot path: `[[unlikely]]`/cold attributes for error paths, PGO or BOLT for layout. 4. Pre-fault and lock memory, use huge pages for large tables. 5. Isolate the core; consider Cache Allocation Technology if other workloads on the socket evict the L3.

And I'd confirm it first: timestamp the order path in production and plot latency against time since the previous order. A cold-path problem shows up as a strong correlation.

4. “A microbenchmark says this function takes 2ns, but in production the same code takes 200ns. What's going on?”

Answer:

Several likely causes, roughly in order: 1. **The benchmark measured nothing.** The result wasn't used, or inputs were constants, and the compiler removed or pre-computed the work. Check the assembly, use `DoNotOptimize`. 2. **Throughput vs latency.** Independent iterations overlap in the pipeline; production calls it once, and the caller waits for the result. Measure with a dependency chain. 3. **Hot vs cold.** In the benchmark, code and data sit in L1 and every branch is predicted. In production, the function runs between other work, so it misses in caches and mispredicts. 200ns is about two cache misses to DRAM. 4. **Different inputs.** Synthetic inputs may take a fast path that real data rarely hits. 5. **Measurement in production.** Timer overhead, or the measurement includes something else (a queue hand-off to another core costs ~100ns by itself).

To resolve it: measure in production with TSC timestamps and hardware counters (`perf stat` on the real process, cache misses and branch misses per call), replay real data in the benchmark, and compare cold-cache and warm-cache runs.