When Nanoseconds Matter:

Ultrafast Trading Systems in C++

DAVID GROSS

Cppcon

The C++ Conference

2024

September 15 - 20

Introduction

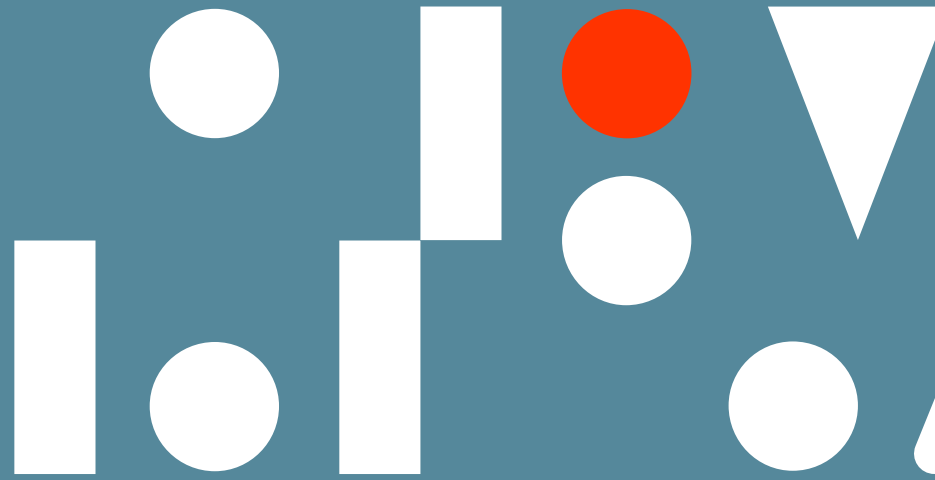
- About me
  - Leading the Options Automated Trading Systems team at Optiver, a global market maker company
  - Worked on low-latency systems for 15 years in Trading and Defense

Introduction

- About me
  - Leading the Options Automated Trading Systems team at Optiver, a global market maker company
  - Worked on low-latency systems for 15 years in Trading and Defense

- Today’s talk
  - Engineering low-latency trading systems
  - Profiling techniques to find bottlenecks
  - Some principles along the way

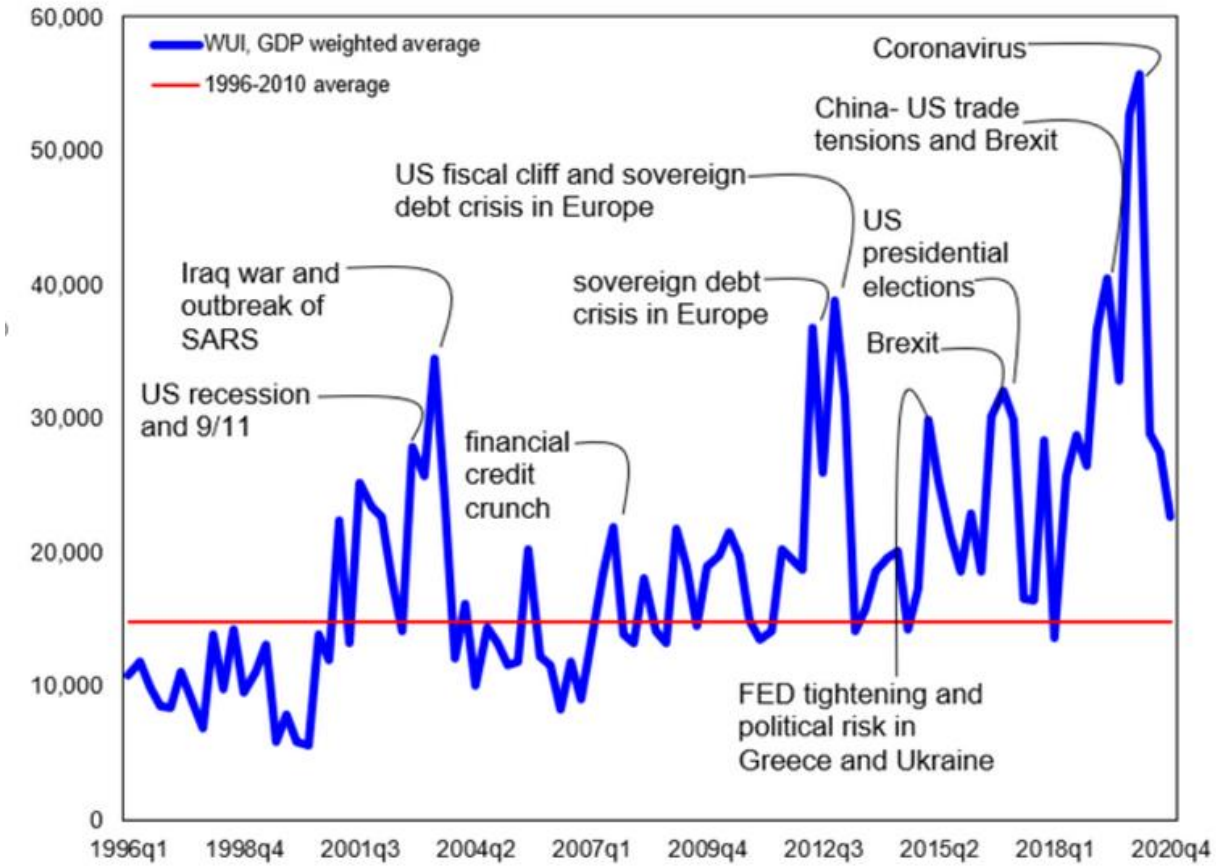
“All that we know, all that we are, began in the depths of antiquity.”
— John Milton



Urban food supply, military campaign… and festivals!

Source: Atlas van Loon

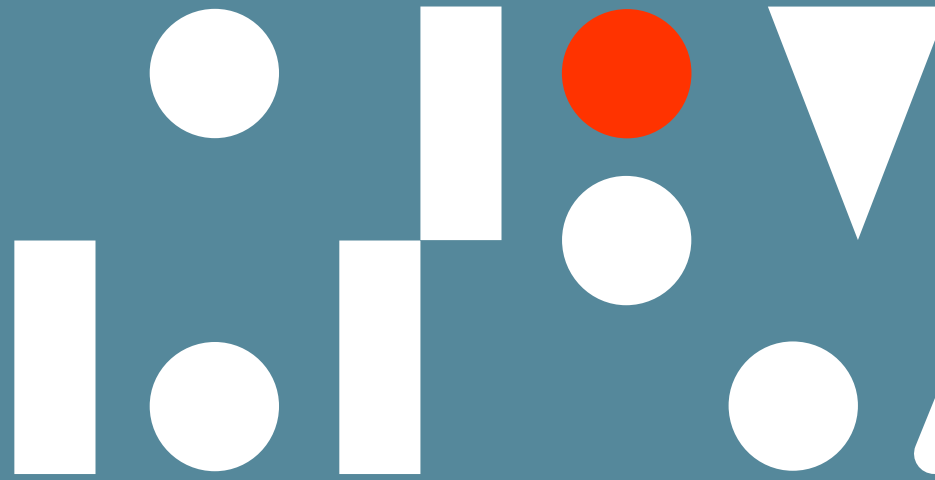
Protecting ourselves against the world's uncertainty



Source: International Monetary Fund (IMF)

Market Making

“There is no silver bullet that will help you beat the market. Successful trading is a losers’ game: it is about being consistently good at everything.”
Charles D. Ellis



Market Making – A Losers' game

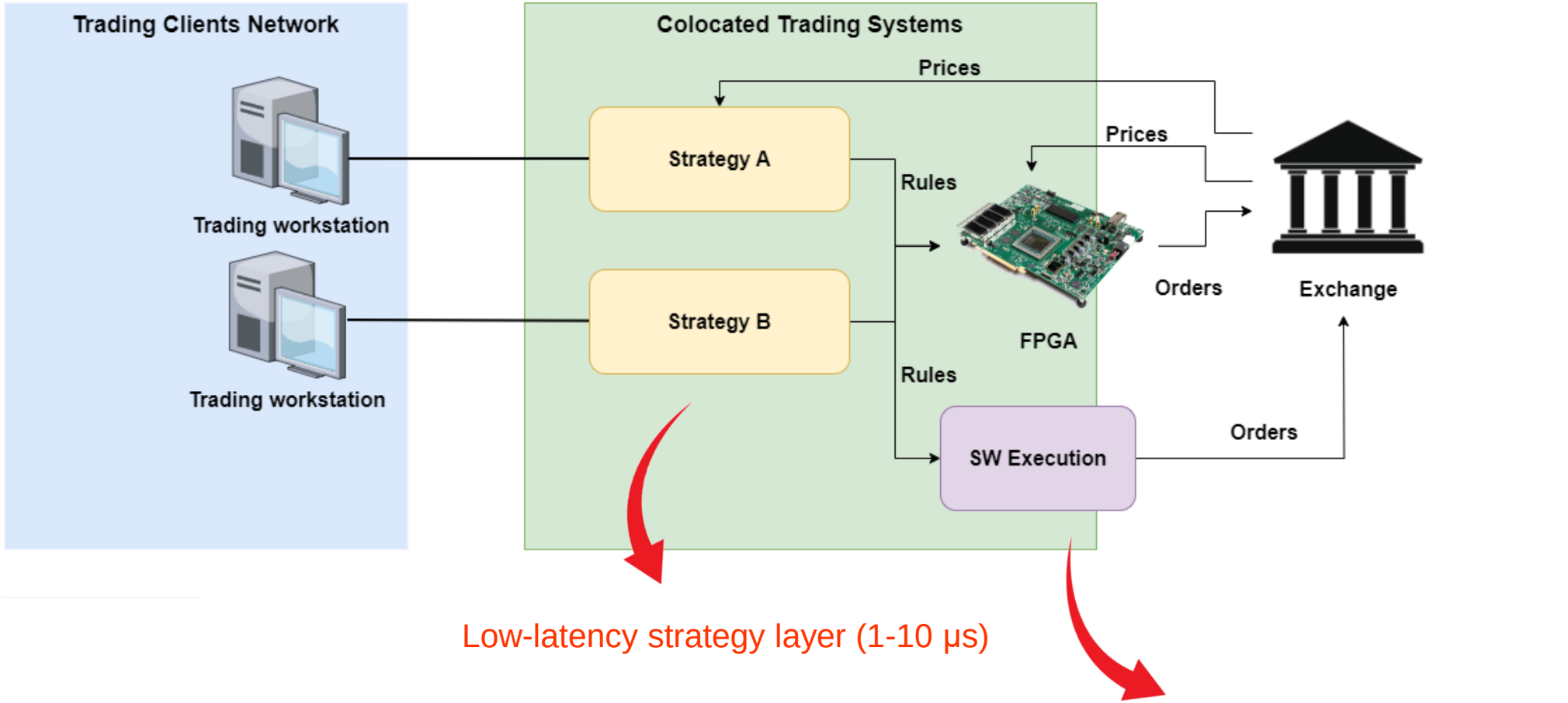
- Be in the market at all times
  - Make small profits from the premium investors pay for you to warehouse risk
  - Avoid big losses

Market Making – A Losers' game

- Be in the market at all times
  - Make small profits from the premium investors pay for you to warehouse risk
  - Avoid big losses

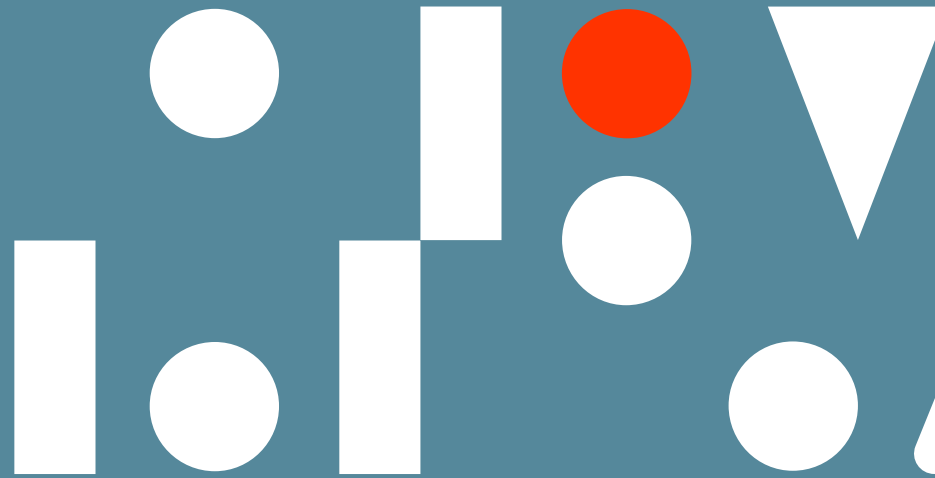
- Combination of
  - Being "fast": you need to *react* "quickly enough" to market events
  - Being "smart": you need the *correct* prices – which is also inherently related to being *fast*

Being fast to get the trades – and to be accurate!

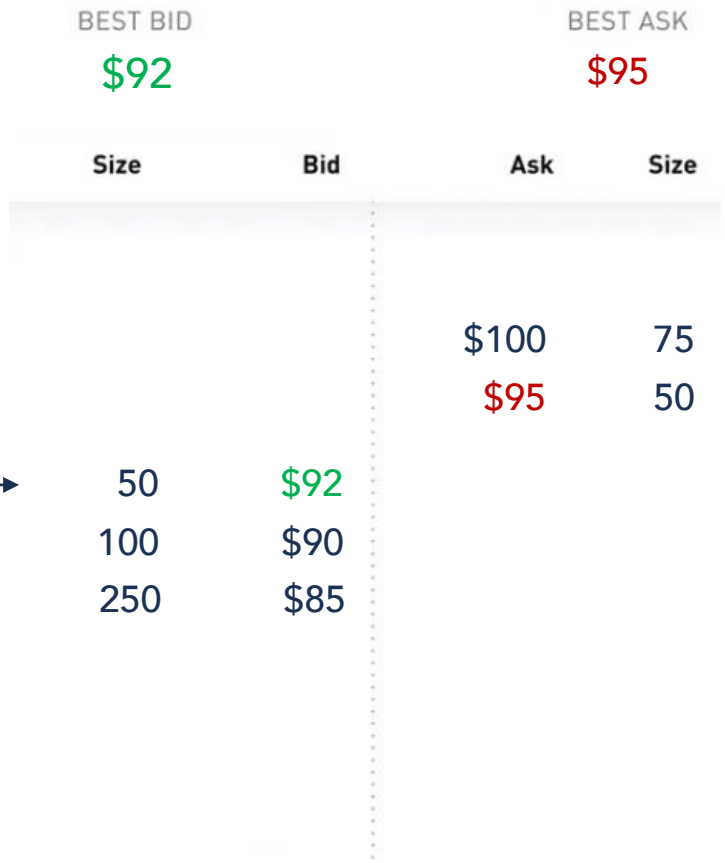


Order book deep dive

“The order book is the heart of any trading system.”
Larry Harris



Order Book In a Nutshell



the “best price” at which you can sell is $92

the “best price” at which you can buy is $95

Order Book – Challenges

- Latency constraint for algorithmic trading
  - Remember – it's not just about being *fast* to send an order, but being *fast* to be accurate

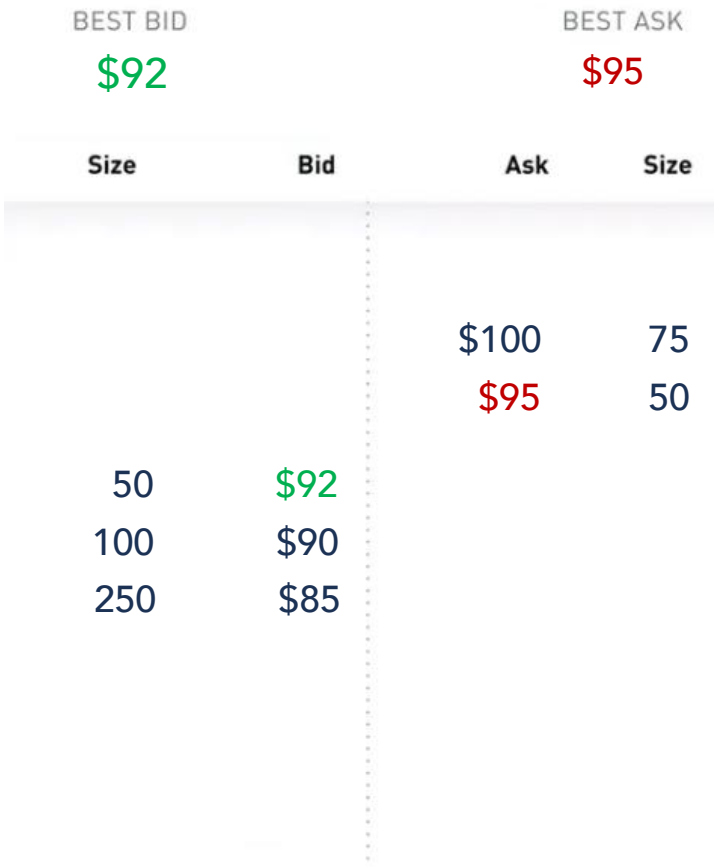
- Latency constraint to not drop packets on the NIC
  - Limited number of buffers

  - Not reading fast enough will cause the application to drop packets, causing an outage

- Up to hundreds of thousands of price updates per second

Order Book – Properties

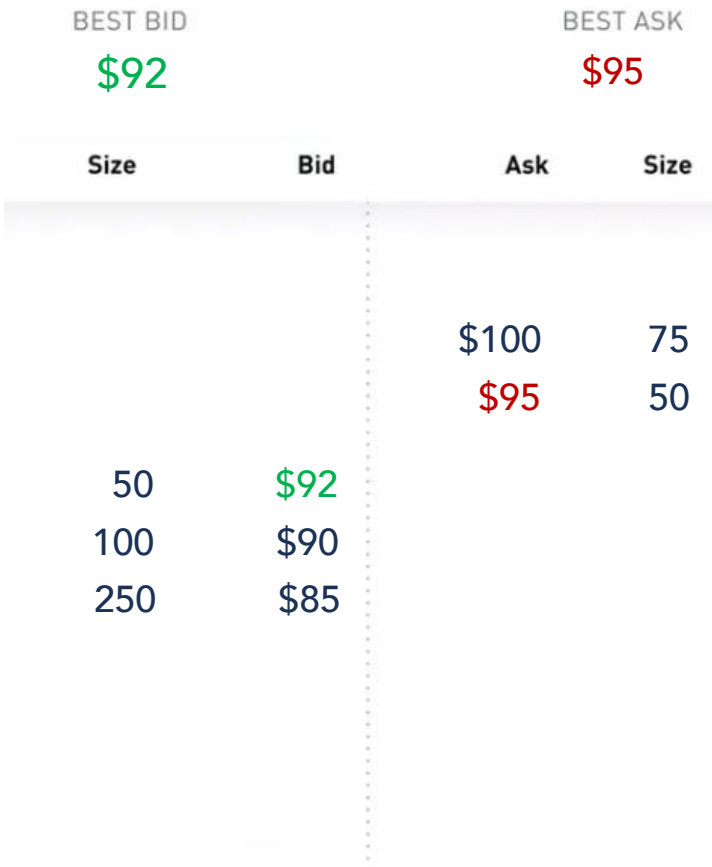
- Two ordered sequences
  - Bids: from highest to lowest price
  - Asks: from lowest to highest price



Order Book – Properties

- Two ordered sequences
  - Bids: from highest to lowest price
  - Asks: from lowest to highest price

- Price Level
  - Price
  - Size: sum of all orders' size at this price level

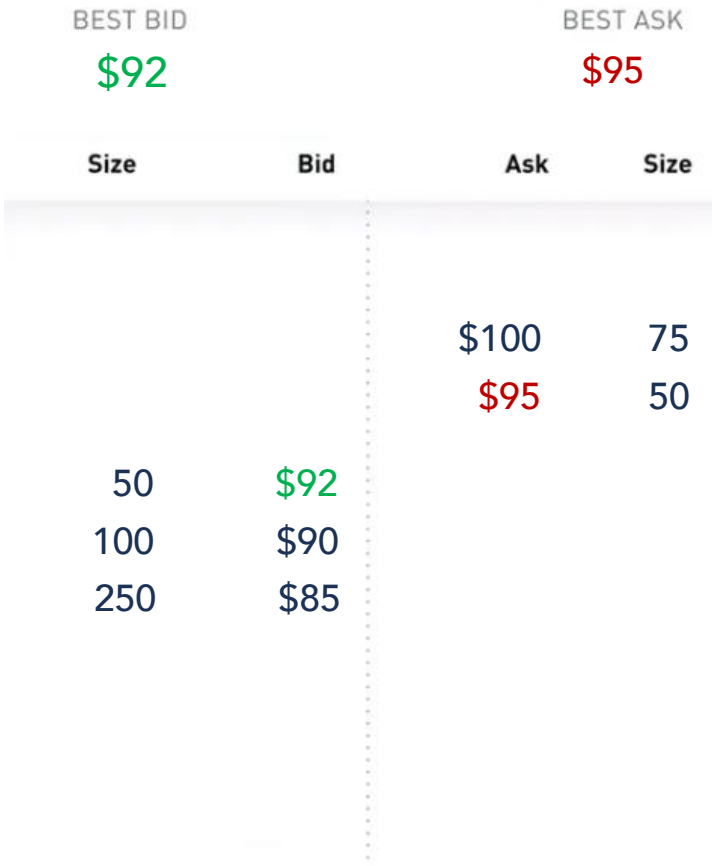


Order Book – Properties

- Two ordered sequences
  - Bids: from highest to lowest price
  - Asks: from lowest to highest price

- Price Level
  - Price
  - Size: sum of all orders' size at this price level

- Each Order has an ID (uint64_t) which is unique throughout the trading session (day)



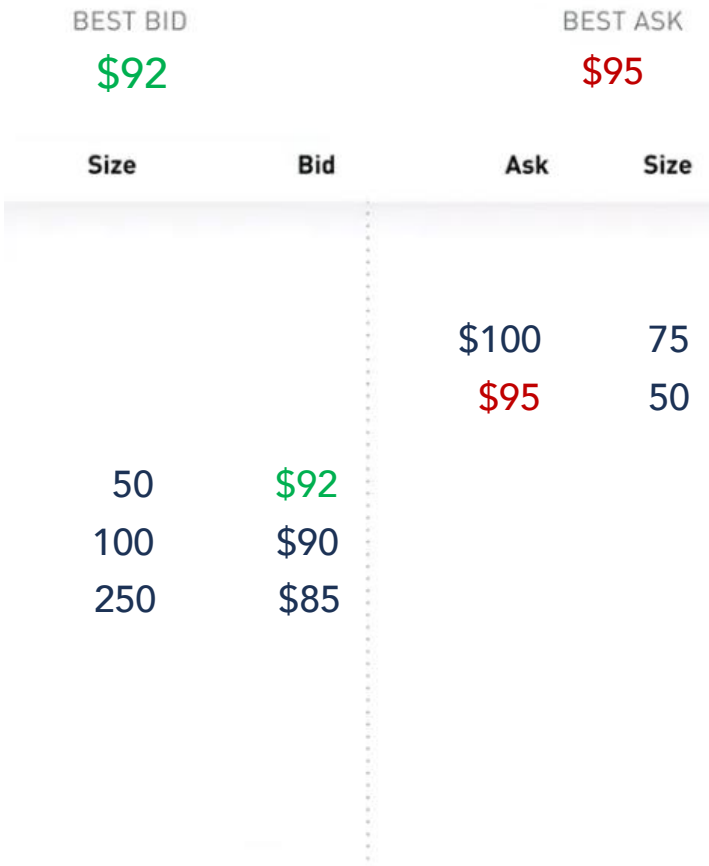
Order Book – Properties

- Two ordered sequences
  - Bids: from highest to lowest price
  - Asks: from lowest to highest price

- Price Level
  - Price
  - Size: sum of all orders' size at this price level

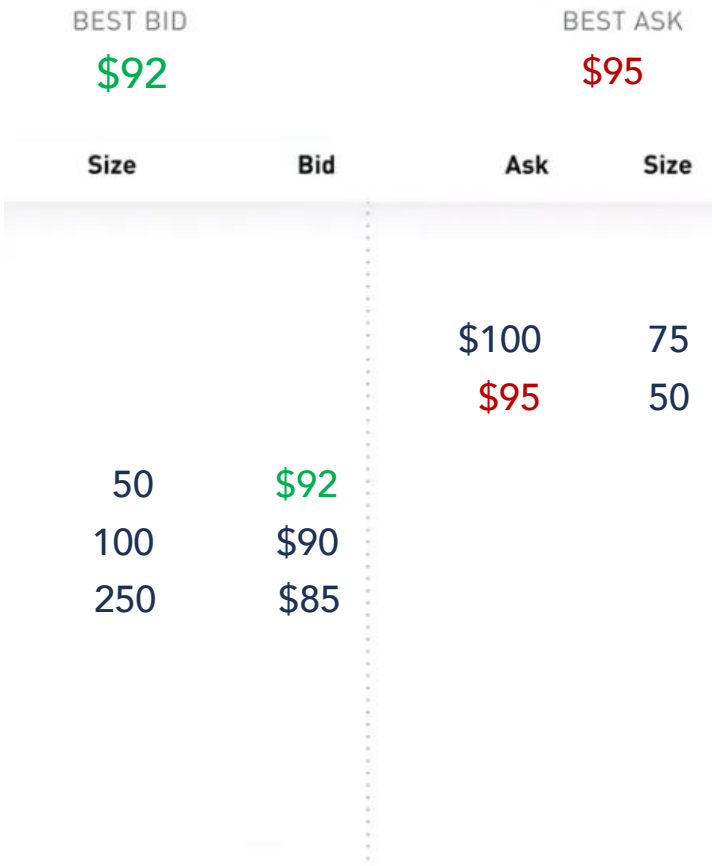
- Each Order has an ID (uint64_t) which is unique throughout the trading session (day)

- A typical stock order book has ~1000 price levels "per side"



Order Book – Exchange messages

```
enum class Side { Bid, Ask };

// in reality, those would be strong types...
using OrderId = uint64_t;
using Volume = int64_t;
using Volume = int64_t;

void AddOrder(OrderId orderId, Side side,
              Price price, Volume volume);


void ModifyOrder(OrderId orderId, Volume newVolume);


void DeleteOrder(OrderId orderId);
```

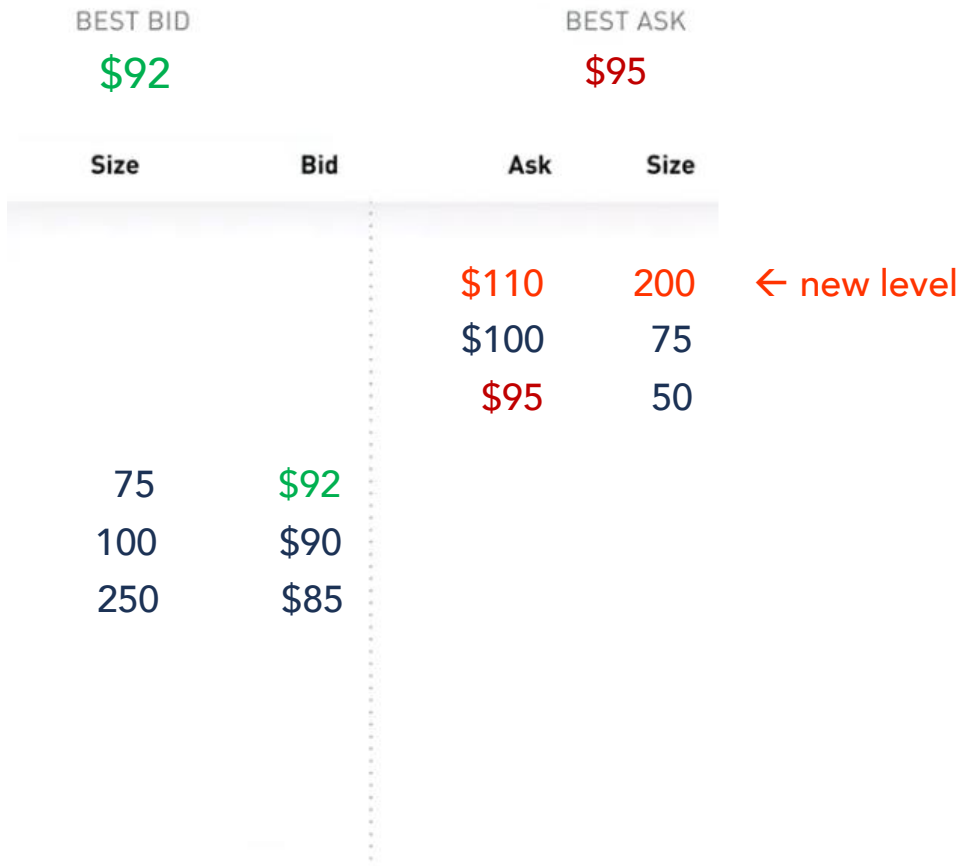


Order Book – Exchange messages

```
AddOrder(OrderId{1}, Side::Buy, Price{92}, Volume{25});
```

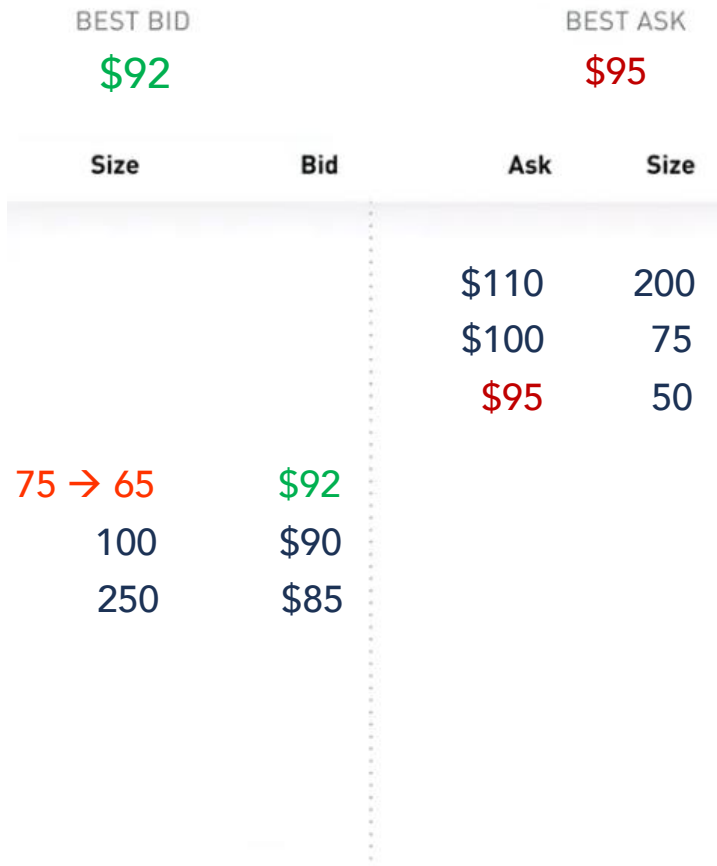
Order Book – Exchange messages

```
AddOrder(OrderId{1}, Side::Buy, Price{92}, Volume{25});

AddOrder(OrderId{2}, Side::Sell, Price{110}, Volume{200});
```



Order Book – Exchange messages

```
AddOrder(OrderId{1}, Side::Buy, Price{92}, Volume{25});

AddOrder(OrderId{2}, Side::Sell, Price{110}, Volume{200});

ModifyOrder(OrderId{1}, Volume{15});
```



Order Book – Exchange messages

```
AddOrder(OrderId{1}, Side::Buy, Price{92}, Volume{25});

AddOrder(OrderId{2}, Side::Sell, Price{110}, Volume{200});

ModifyOrder(OrderId{1}, Volume{15});

DeleteOrder(OrderId{2});
```

Order Book – Initial Observation

- No matter what data structure we choose for our Order Book – we need a hash table

```
void AddOrder(uint64_t orderId, Side side,
              Price price, Volume volume)
{
    ...

    auto [it, inserted] = mOrders.emplace(orderId, ...);
    EXPECT(inserted, “duplicate order”);

    ...
}
```

```
void DeleteOrder(uint64_t orderId)
{
    auto it = mOrders.find(orderId);
    EXPECT(it != mOrders.end(), “missing order”);

    ...
}
```

Order Book – First Take

```
std::map<Price, Volume, std::greater<Price>> mBidLevels;
std::map<Price, Volume, std::less<Price>> mAskLevels;
```

Order Book – First Take

```
std::map<Price, Volume, std::greater<Price>> mBidLevels;
std::map<Price, Volume, std::less<Price>> mAskLevels;

template <class T>
typename T::iterator AddOrder(T& levels, Price price, Volume volume)
{
    auto [it, inserted] = levels.try_emplace(price, volume);
    if (inserted == false)
        it->second += volume;
    return it;
}
```

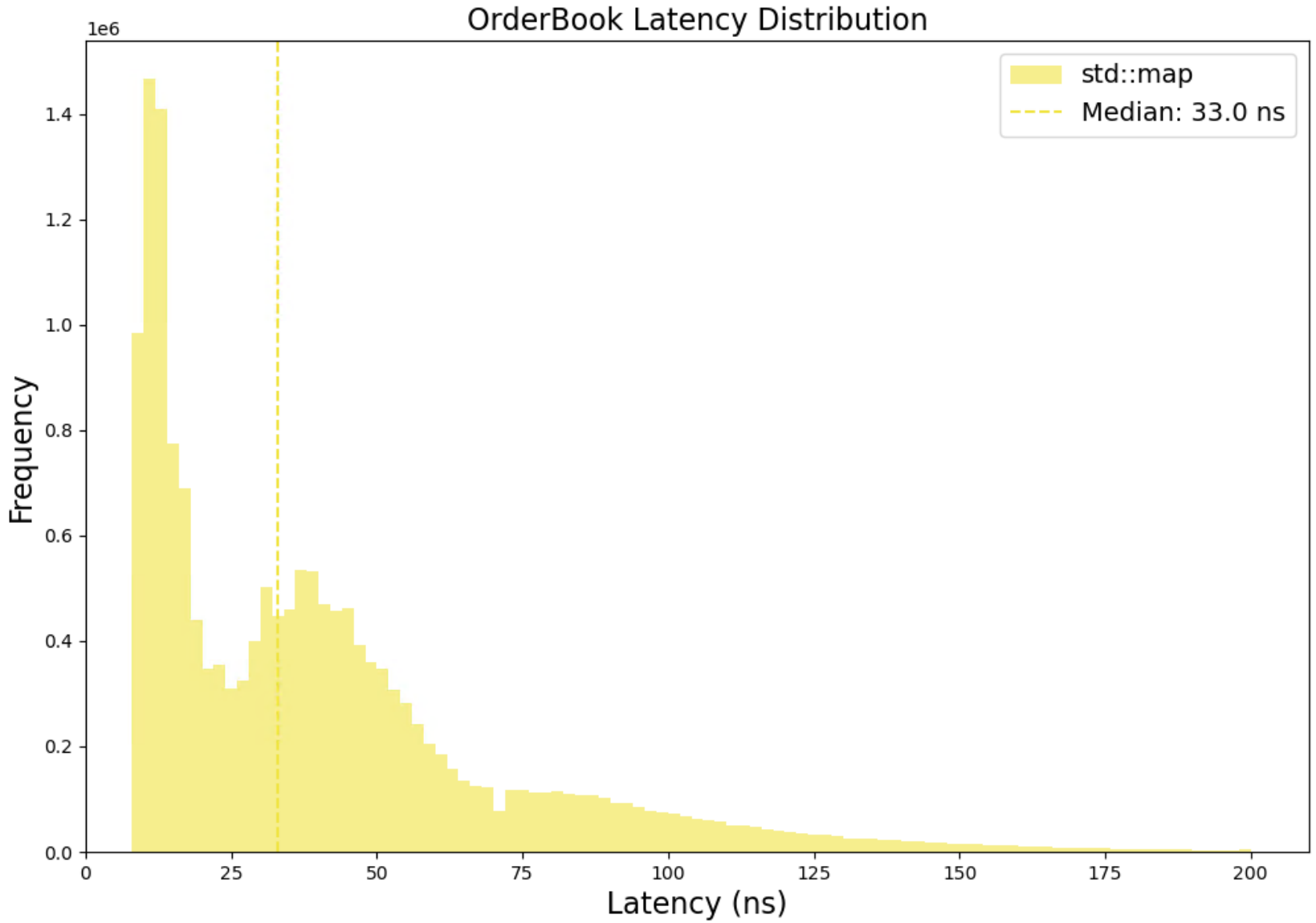
Order Book – First Take

```
std::map<Price, Volume, std::greater<Price>> mBidLevels;
std::map<Price, Volume, std::less<Price>> mAskLevels;

template <class T>
typename T::iterator AddOrder(T& levels, Price price, Volume volume)
{
    auto [it, inserted] = levels.try_emplace(price, volume);
    if (inserted == false)
        it->second += volume;
    return it;
}

template <class T>
void DeleteOrder(typename T::iterator it, T& levels, Price price, Volume volume)
{
    it->second -= volume;
    if (it->second <= 0)
        levels.erase(it);
}
```

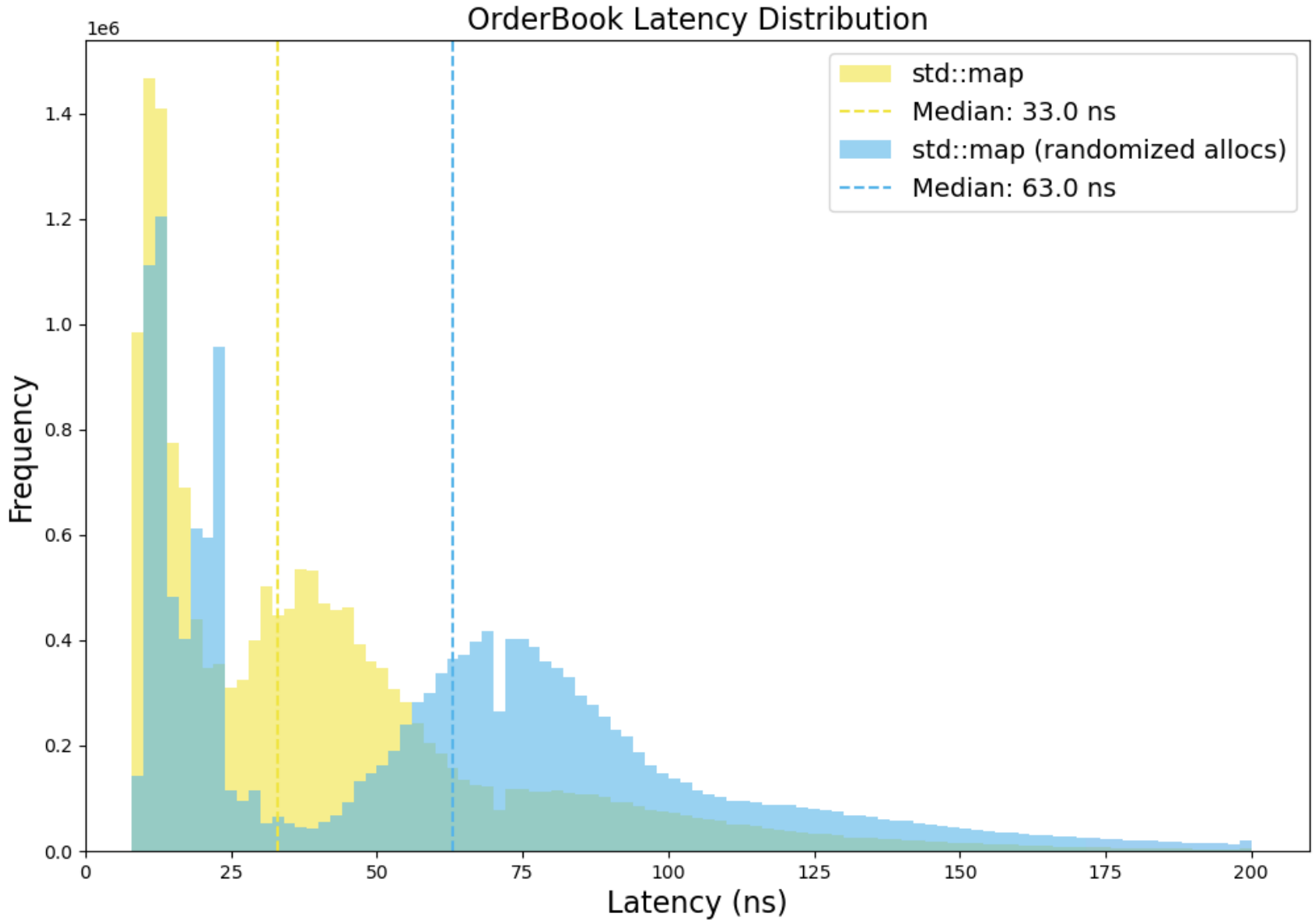
OrderBookMap – Complexity

- AddOrder: *log(N)*
- ModifyOrder: *amortized constant*
  - std::map iterators are stable, which means we can store it with our order data (in the hash table)
- DeleteOrder: *amortized constant*
  - Same as above

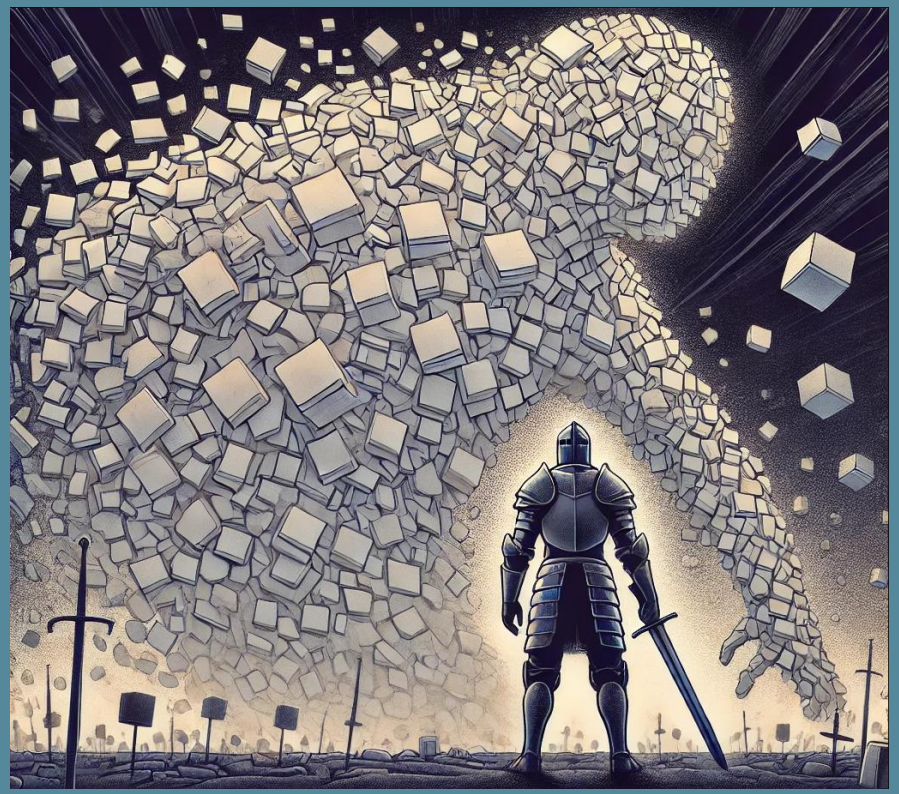
OrderBookMap Latencies



OrderBookMap Latencies – The True Story



Principle #1: “Most of the time, you don’t want node containers”



Source: ChatGPT

Using std::vector

- Using two vectors (bids and asks) and use *std::lower_bound* (binary search)

```
std::vector<std::pair<Price, Volume>> mBidLevels;
std::vector<std::pair<Price, Volume >> mAskLevels;
```

Using std::vector

- Using two vectors (bids and asks) and use *std::lower_bound* (binary search)

```
std::vector<std::pair<Price, Volume>> mBidLevels;
std::vector<std::pair<Price, Volume >> mAskLevels;
```

- AddOrder:
  - *log(N)* if the price level exists
  - *log(N)* + *N* if a new price level is inserted
- ModifyOrder: *log(N)*
  - We can’t store iterators/pointers as they are being invalidated on a call to *std::vector::insert*
- DeleteOrder:
  - *log(N)* if the price level exists
  - *log(N)* + *N* if a new price level is inserted

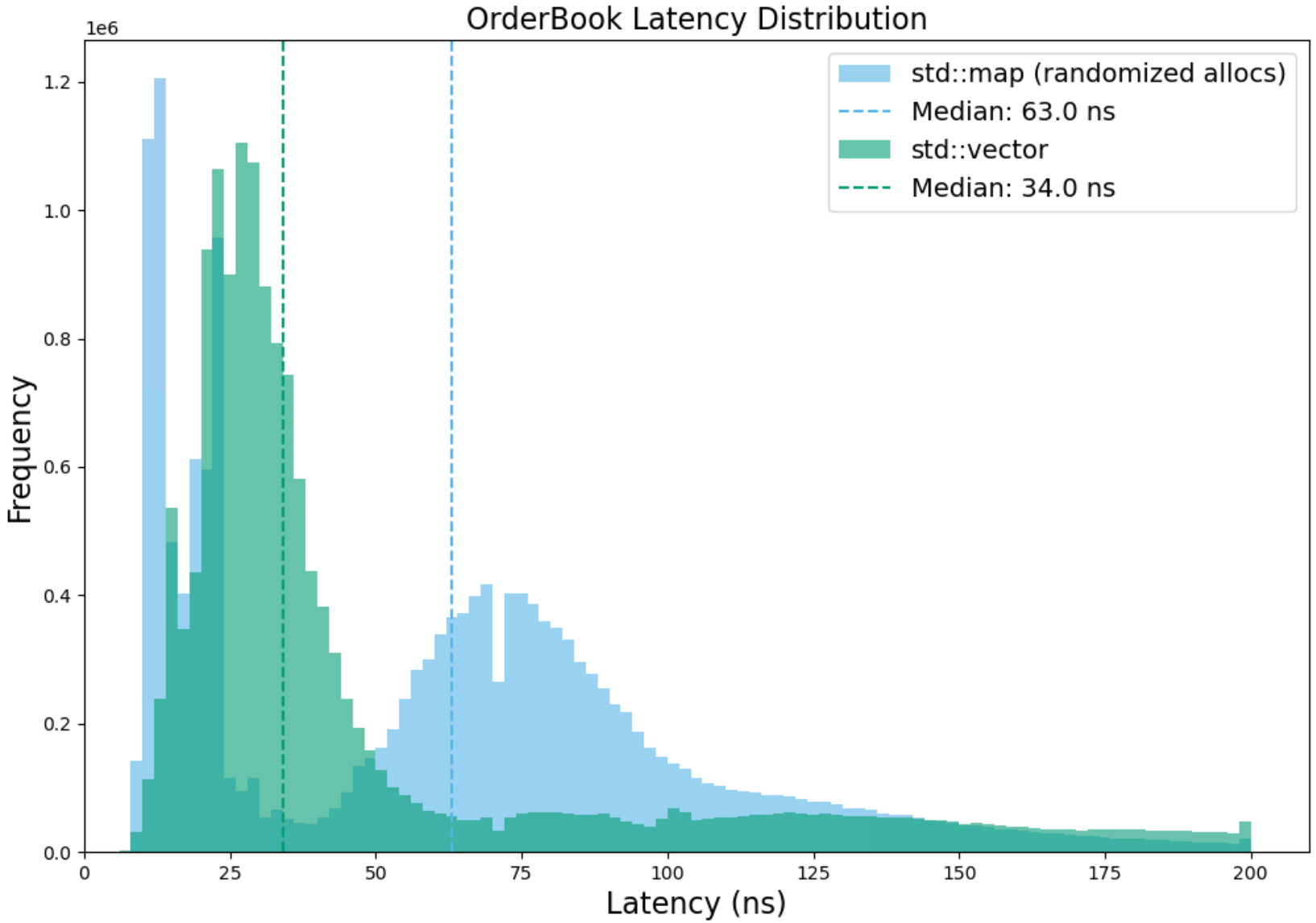
Order Book – Vector

```
void AddOrder(Side side, Price price, Volume volume)
{
    if (side == Side::Bid)
        return AddOrder(bidLevels, price, volume, std::greater<Price>());
    else
        return AddOrder(askLevels, price, volume, std::less<Price>());
}
```

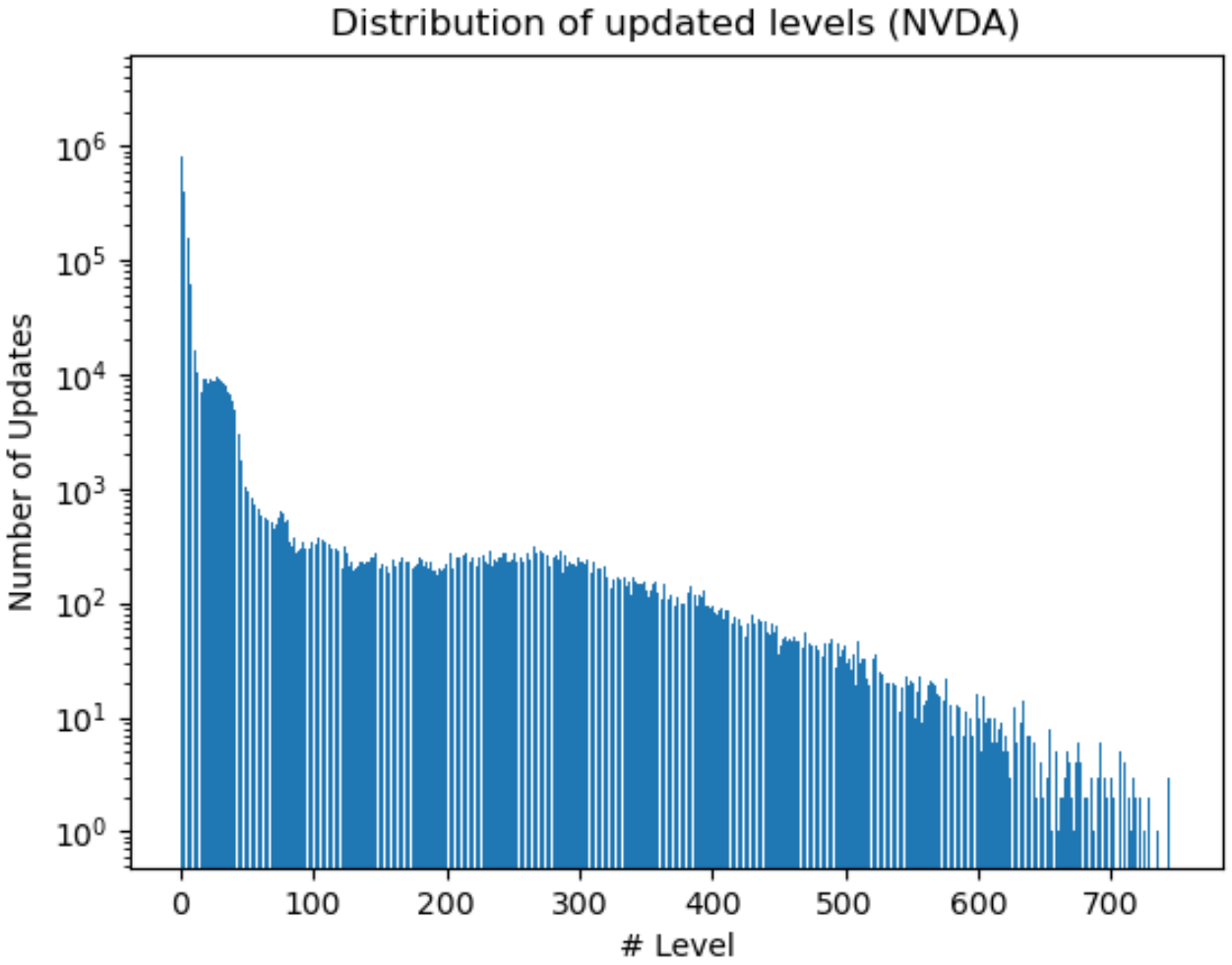
Order Book – Vector

```
void AddOrder(Side side, Price price, Volume volume)
{
    if (side == Side::Bid)
        return AddOrder(bidLevels, price, volume, std::greater<Price>());
    else
        return AddOrder(askLevels, price, volume, std::less<Price>());
}

template <class T, class Compare>
void AddOrder(T& levels, Price price, Volume volume, Compare comp)
{
    auto it = std::lower_bound(levels.begin(), levels.end(), price,
              [comp](const auto& p, Price price) { return comp(p.first, price); });

    if (it != levels.end() && it->first == price)
        it->second += volume;
    else
        levels.insert(it, {price, volume});
}
```

std::map vs std::vector



A brief look at our data



Order Book – Vector

```
void AddOrder(Side side, Price price, Volume volume)
{
    if (side == Side::Bid)
        return AddOrder(bidLevels, price, volume, std::greater<int64_t>());
    else
        return AddOrder(askLevels, price, volume, std::less<int64_t>());
}
```

```
bids = {92, 90, 85}
asks = {95, 100}
```

BEST BID		BEST ASK	
$92			$95
Size	Bid	Ask	Size
		$100	75
		$95	50
50	$92		
100	$90		
250	$85		

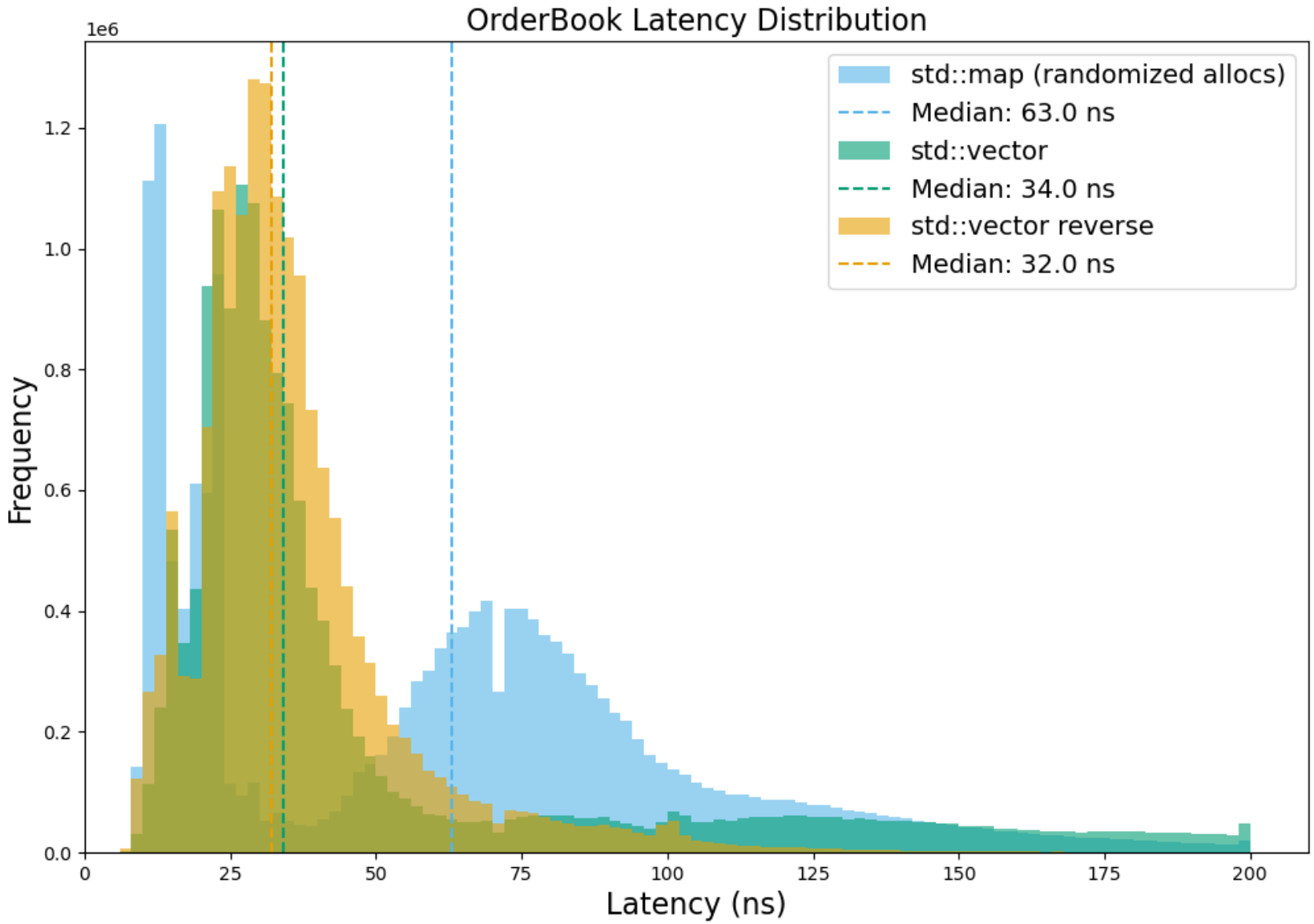
```
auto GetBestPrices() const
{
    EXPECT(!mBidLevels.empty() && !mAskLevels.empty());
    return {mBidLevels.begin().first, mAskLevels.begin().first};
}
```

Order Book – “reverse” Vector

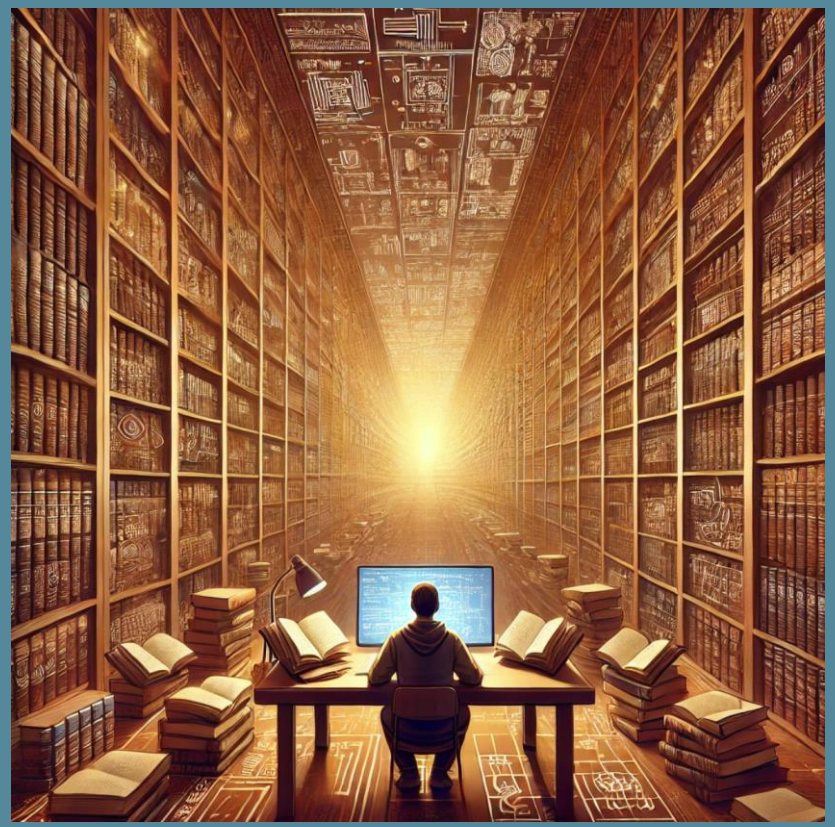
- “Counter-intuitive” ordering – our “best price” or “top” is at the end of the collection…
- … But we minimize our number of copies!

```
void AddOrder(Side side, Price price, Volume volume)
{
    if (side == Side::Bid)
        return AddOrder(bidLevels, price, volume, std::less<Price>());
    else
        return AddOrder(askLevels, price, volume, std::greater<Price>());
}


auto GetBestPrices() const
{
    return {mBidLevels.rbegin().first, mAskLevels.rbegin().first};
}
```

std::map vs std::vector



Principle #2: “Understanding your problem (by looking at data!)”



Source: ChatGPT

Principle #3: “Hand tailored (specialized) algorithms are key to achieve performance ”

Source: ChatGPT

Running *perf* on a benchmark

- Can’t run *perf* on the entire binary if our initialization function is “big”

```
void RunPerf()
{
    pid_t pid = fork();
    if (pid == 0)
    {
        const auto parentPid = std::to_string(getppid());
        std::cout << "Running perf on parent process " << parentPid << std::endl;
        execlp("perf", "perf", ..., parentPid.c_str(), (char*)nullptr);
        throw std::runtime_error("execlp failed");
    }
}

void InitAndRunBenchmark()
{
    InitBenchmark(); // might take a long time!
    RunPerf();
    RunBenchmark();
}
```

Running *perf* on a benchmark

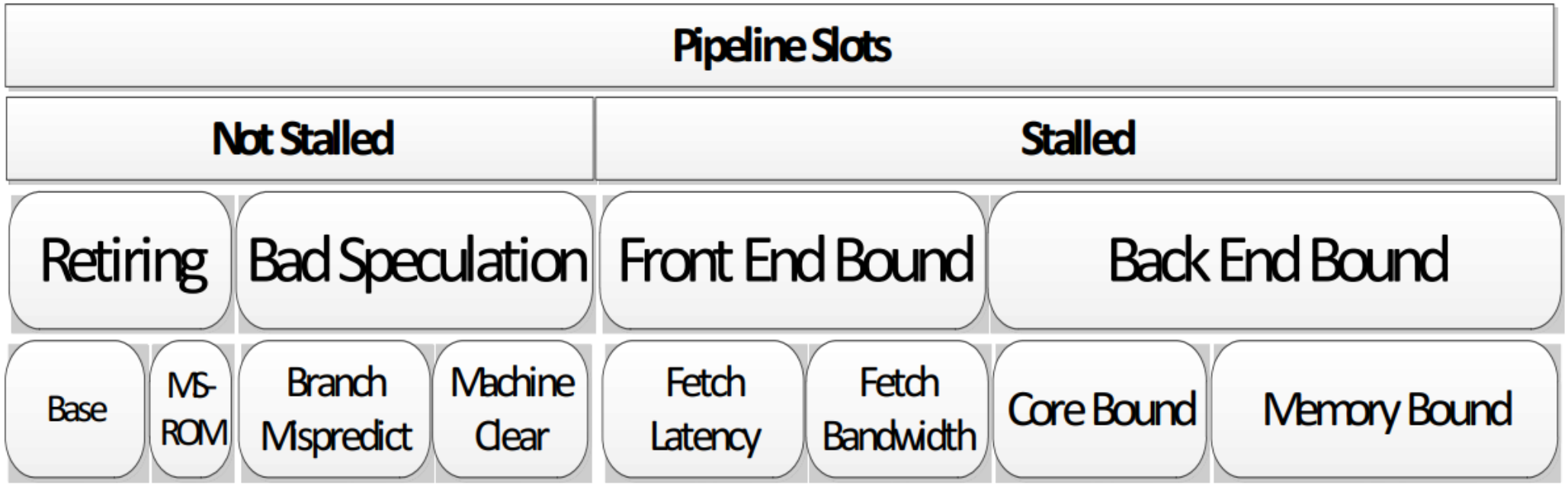
- First *perf* measurements should never be too specific

```
perf stat –I 10000 –M Frontend_Bound,Backend_Bound,Bad_Speculation,Retiring –p pid
```

Running *perf* on a benchmark

- First *perf* measurements should never be too specific

```
perf stat -I 10000 -M Frontend_Bound,Backend_Bound,Bad_Speculation,Retiring -p pid
```



Source: intel.com

Running *perf* on a benchmark

- First *perf* measurements should never be too specific

```
perf stat -I 10000 -M Frontend_Bound,Backend_Bound,Bad_Speculation,Retiring -p pid

# counts         unit                                                  events
43194329517      de_src_op_disp.all:u                    #     25.0 %  bad_speculation
13998678938      ls_not_halted_cyc:u                     #     26.4 %  retiring
22162353234      ex_ret_ops:u
15111861755      de_no_dispatch_per_slot.backend_stalls:u      #     17.9 %  backend_bound
14041133285      ls_not_halted_cyc:u
25801294995      de_no_dispatch_per_slot.no_ops_from_frontend:u #     30.6 %  frontend_bound
14049463430      ls_not_halted_cyc:u
```

perf record –g –p <pid>

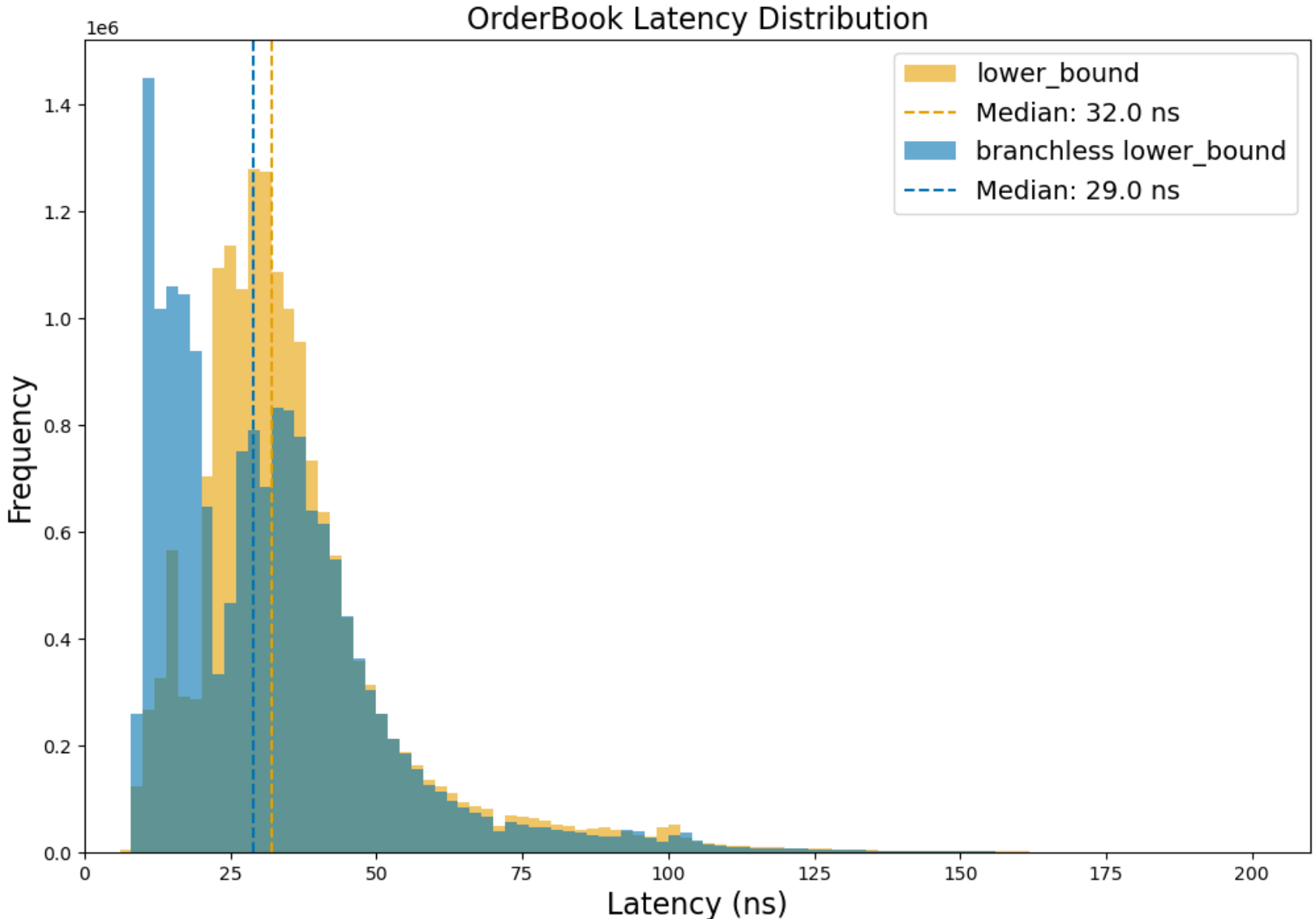
```
Percent | 28:   test   rax,rax
  1.99  |       jle    52
        | 2d:   mov    rcx,rax
  1.40  |       sar    rcx,1
  0.59  |       mov    rdx,rcx
  5.14  |       shl    rdx,0x4
  0.70  |       add    rdx,r9
  0.36  |       cmp    rsi,QWORD PTR [rdx]
 15.61  |       jle    80
        |       sub    rax,rcx
  5.93  |       lea    r9,[rdx+0x10]
  0.12  |       sub    rax,0x1
  0.81  |       test   rax,rax
        |       jg     2d
        | 52:   cmp    r10,r9
  2.57  |       je     60
        |       cmp    rsi,QWORD PTR [r9]
  2.34  |       je     f8
        | 60:   mov    QWORD PTR [rbp-0x10],rsi
  0.82  |       lea    rdx,[rbp-0x10]
        |       mov    rsi,r9
        |       mov    QWORD PTR [rbp-0x8],r8
        |       call   std::vector<std::pair<long, long>, std::allocato
  0.12  |       leave
        |       ret
        |       cs     nop WORD PTR [rax+rax*1+0x0]
        | 80:   mov    rax,rcx
  5.30  |       jmp    28
        |       nop
        | 88:   mov    r10,QWORD PTR [rdi+0x20]
  0.94  |       mov    r9,QWORD PTR [rdi+0x18]
```

```
  0.10  |       mov    rax,r10
  0.47  |       sub    rax,r9
        |       sar    rax,0x4
  0.70  |       nop
        | a0:   test   rax,rax
  1.41  |       jle    ca
        | a5:   mov    rcx,rax
  3.04  |       sar    rcx,1
  0.94  |       mov    rdx,rcx
  5.90  |       shl    rdx,0x4
  0.48  |       add    rdx,r9
  0.23  |       cmp    rsi,QWORD PTR [rdx]
 17.45  |       jge    f0
        |       sub    rax,rcx
  5.16  |       lea    r9,[rdx+0x10]
  0.47  |       sub    rax,0x1
  0.45  |       test   rax,rax
        |       jg     a5
        | ca:   cmp    r10,r9
  2.96  |       je     d4
        |       cmp    rsi,QWORD PTR [r9]
  2.45  |       je     f8
        | d4:   mov    QWORD PTR [rbp-0x10],rsi
  2.11  |       lea    rdx,[rbp-0x10]
        |       add    rdi,0x18
  0.12  |       mov    rsi,r9
        |       mov    QWORD PTR [rbp-0x8],r8
        |       call   std::vector<std::pair<long, long>, std::allocato
        |       leave
        |       ret
        |       xchg   ax,ax
        | f0:   mov    rax,rcx
  5.47  |       jmp    a0
        |       nop
        | f8:   add    QWORD PTR [r9+0x8],r8
  1.87  |       leave
        |       ret
```

Branchless binary search

```
template <class ForwardIt, class T, class Compare>
ForwardIt branchless_lower_bound(ForwardIt first, ForwardIt last, const T& value, Compare comp)
{
    auto length = last - first;
    while (length > 0)
    {
        auto half = length / 2;
        // multiplication (by 1) is needed for GCC to generate CMOV
        first += comp(first[half], value) * (length - half);
        length = half;
    }
    return first;
}
```

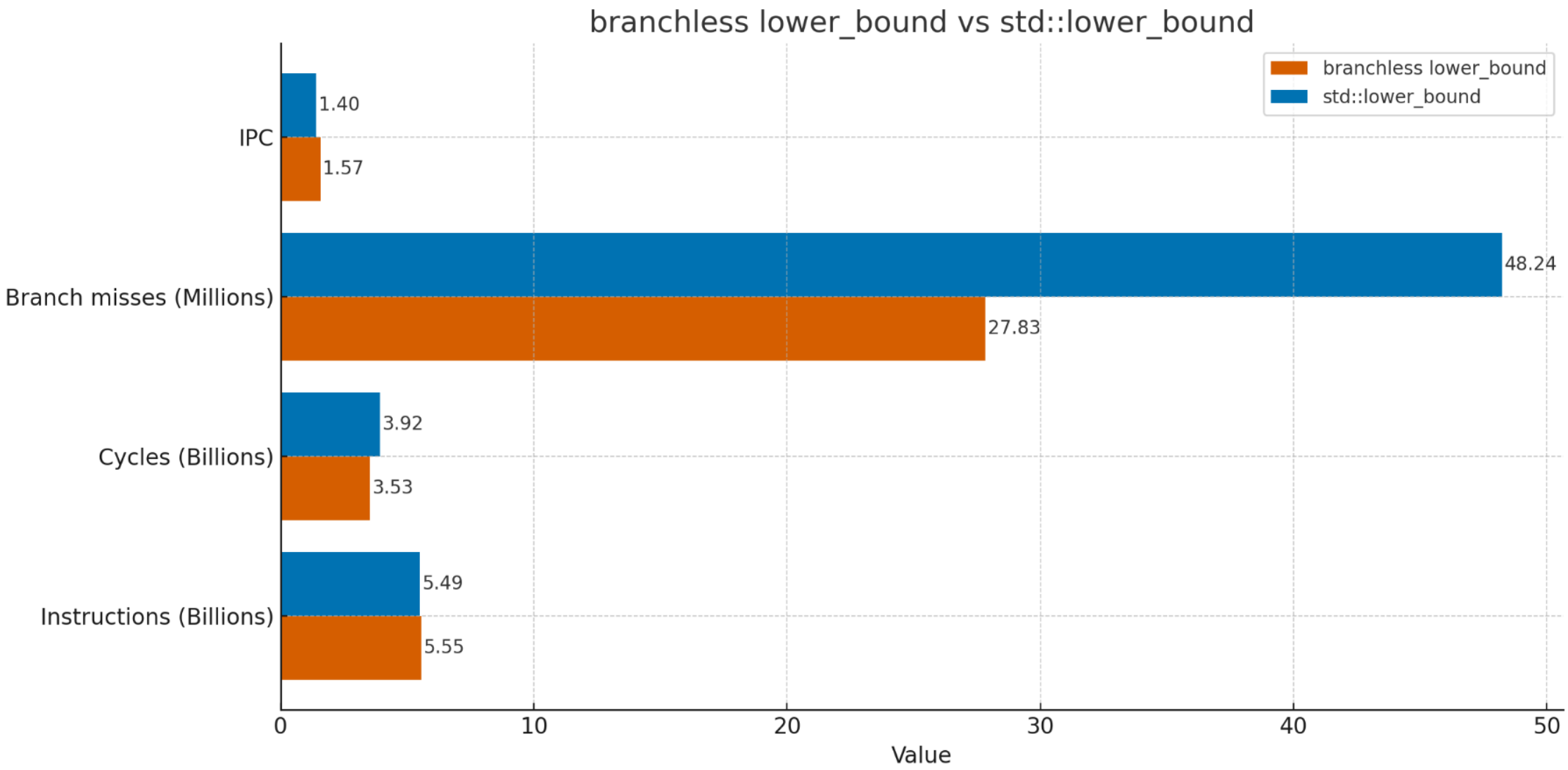
Branchless binary search



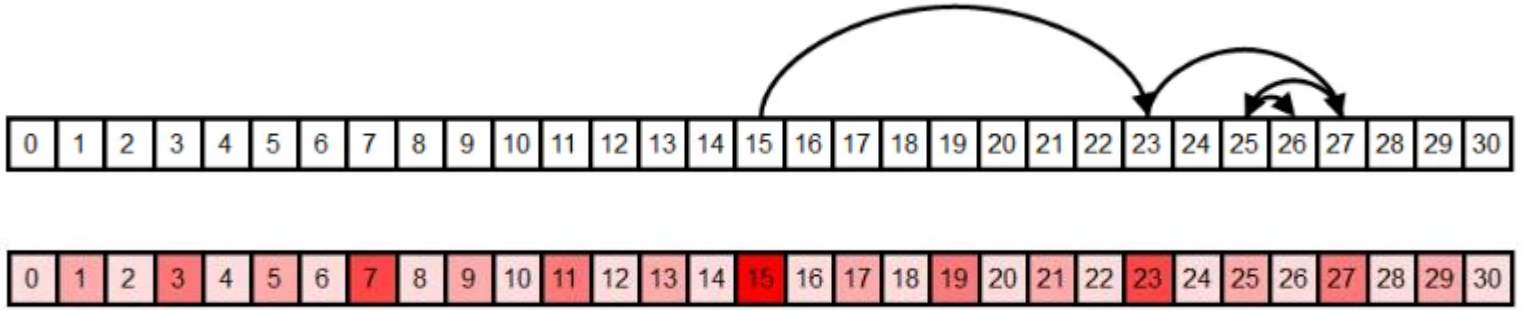
HW counters with libpapi

```
#include "papipp.h“       // see https://github.com/david-grs/papipp/tree/master

papi::event_set<PAPI_TOT_INS, PAPI_TOT_CYC, PAPI_BR_MSP, PAPI_L1_DCM> events;
events.start_counters();

RunBenchmark();

events.stop_counters();

std::cout << events.get<PAPI_TOT_INS>().counter() << " instructions" << std::endl;
std::cout << events.get<PAPI_TOT_CYC>().counter() << " cycles" << std::endl;
std::cout << events.get<PAPI_L1_DCM>().counter() << " d1 cache misses" << std::endl;
std::cout << events.get<PAPI_BR_MSP>().counter() << " branch misses" << std::endl;
std::cout << "IPC = "
          << (events.get<PAPI_TOT_INS>().counter() / (double)events.get<PAPI_TOT_CYC>().counter())
          << std::endl;
```

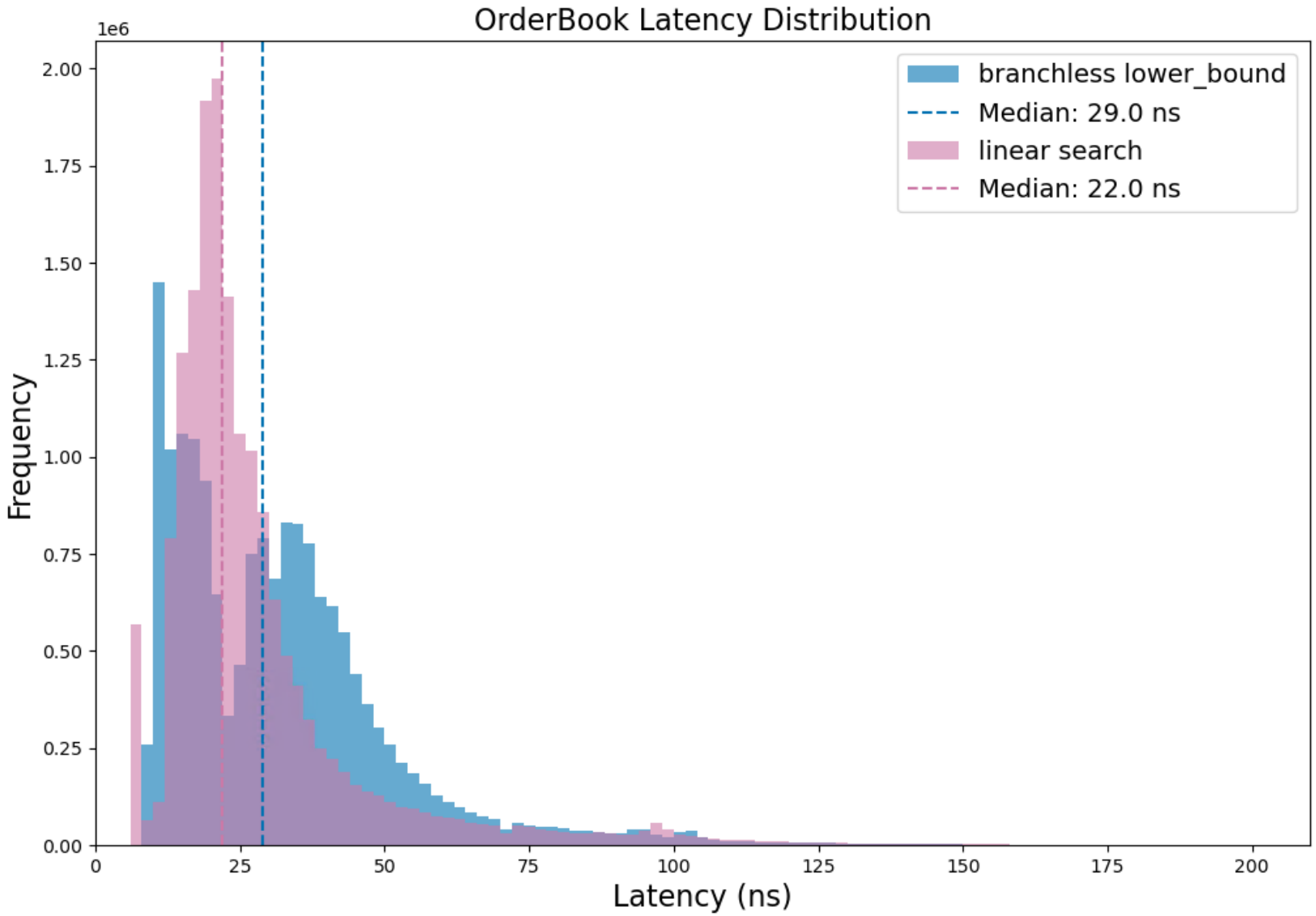
Branchless binary search



Binary search – memory access



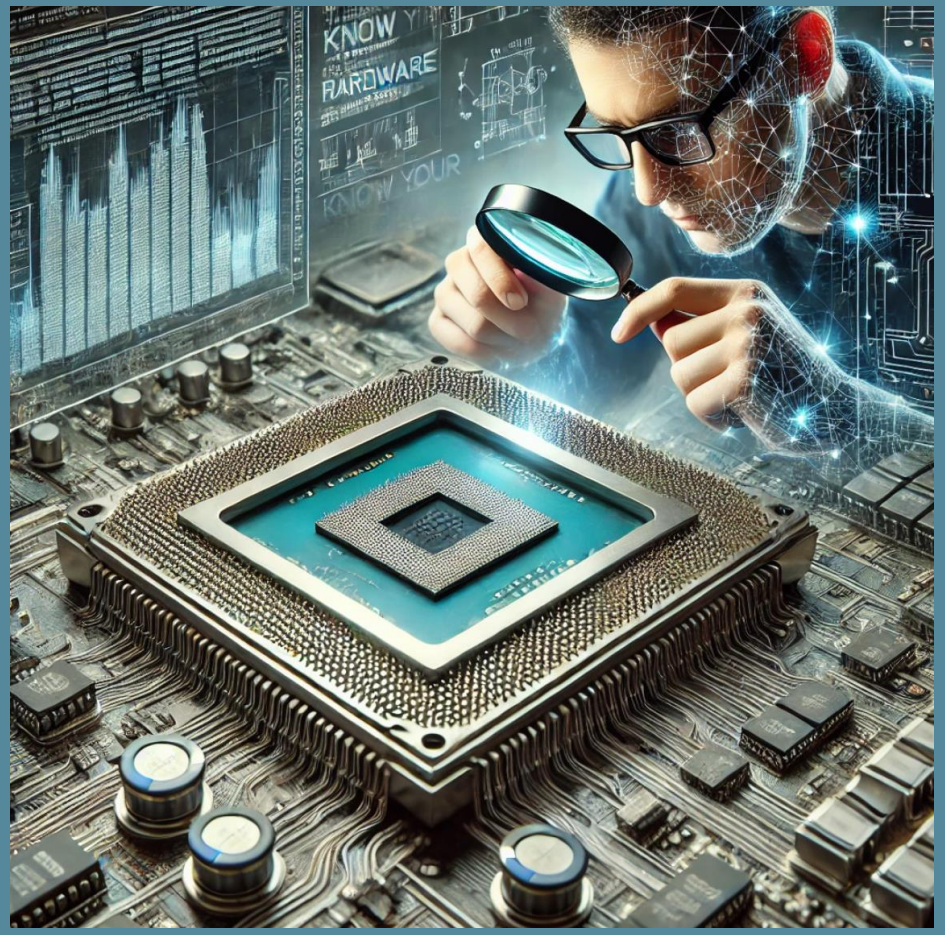
Linear search



Principle #4: “Simplicity is the ultimate sophistication”

Source: ChatGPT

Principle #5: “Mechanical sympathy”



Source: ChatGPT

I-Cache misses – likely/unlikely attributes

```
template <class T, class Compare>
void AddOrder(T& levels, Price price, Volume qty, Compare comp)
{
    auto it = std::ranges::find_if(levels.begin(), levels.end(), price,
              [comp](const auto& p, Price price) { return comp(p.first, price); });

    if (it != levels.end() && it->first == price) [[likely]]
    {
        it->second += volume;
    }
    else
    {
        levels.insert(it, {price, volume});
    }
}
```

I-Cache misses – likely/unlikely attributes

```
template <class T, class Compare>
void AddOrder(T& levels, Price price, Volume qty, Compare comp)
{
    auto it = std::ranges::find_if(levels.begin(), levels.end(), price,
              [comp](const auto& p, Price price) { return comp(p.first, price); });

    if (it != levels.end() && it->first == price) [[likely]]
    {
        it->second += volume;
    }
    else
    {
        levels.insert(it, {price, volume});
    }
}
```

```
.L2:
        cmp     rbx, r15
        je      .L4
        cmp     DWORD PTR [rbx], ebp
        jne     .L5
        add     DWORD PTR [rbx+4], edx
.L1:
        add     rsp, 24
        pop     rbx
        pop     rbp
```

I-Cache misses – Immediately Invoked Function Expressions (IIFE)

```
template <class T, class Compare>
void DeleteOrder(T& levels, Price price, Volume volume, Compare comp)
{
    auto it = std::ranges::find_if(levels.begin(), levels.end(), price,
              [comp](const auto& p, Price price) { return comp(p.first, price); });

    EXPECT(it != levels.end() && it->first == price);

    it->second -= volume;
    if (it->second <= 0)
    {
        levels.erase(it);
    }
}
```

I-Cache misses – Immediately Invoked Function Expressions (IIFE)

```
template <class T, class Compare>
void DeleteOrder(T& levels, Price price, Volume volume, Compare comp)
{
    auto it = std::ranges::find_if(levels.begin(), levels.end(), price,
              [comp](const auto& p, Price price) { return comp(p.first, price); });

    EXPECT(it != levels.end() && it->first == price);

    it->second -= volume;
    if (it->second <= 0)
    {
        levels.erase(it);
    }
}
```

```
if (!(it != levels.end() && it->first == price)) [[unlikely]]
{
    [&]() __attribute__((noinline, cold)) { HandleError(); }();
}
```

I-Cache misses – Immediately Invoked Function Expressions (IIFE)

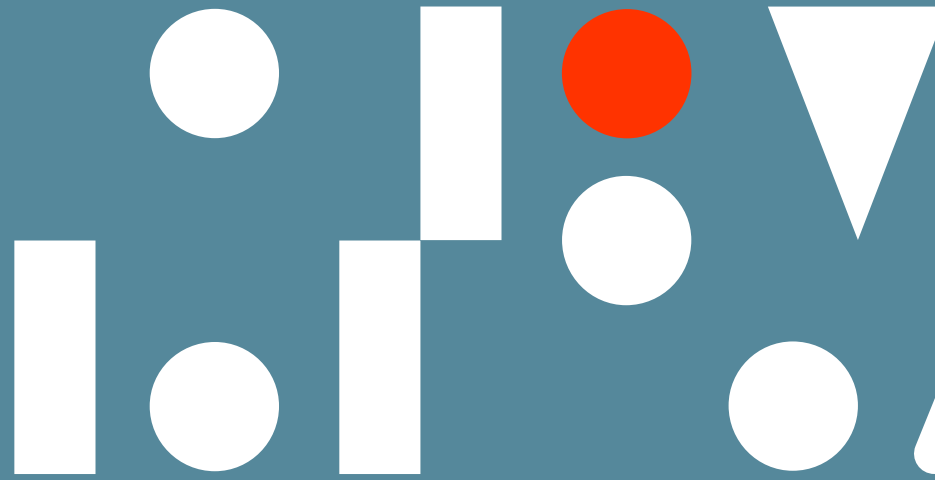
```
template <class T, class Compare>
void DeleteOrder(T& levels, Price price, Volume volume, Compare comp)
{
    auto it = std::ranges::find_if(levels.begin(), levels.end(), price,
              [comp](const auto& p, Price price) { return comp(p.first, price); });

    EXPECT(it != levels.end() && it->first == price);

    it->second -= volume;
    if (it->second <= 0)
    {
        levels.erase(it);
    }
}
```

```
DeleteOrder(std::vector<std::pair<int, int>, std::allocator<std::pair<int, int> > >&, int, int) [clone .cold]:
.L5:
        mov     DWORD PTR [rsp+12], edx
        call    DeleteOrder(std::vector<std::pair<int, int>, std::allocator<std::pair<int, int> > >&, int, int):
        mov     edx, DWORD PTR [rsp+12]
        jmp     .L6
```

Lambda, Functor vs std::function

```
OrderBookVector::OrderBookVector() :
    mBidsCompare([](const std::pair<Price, Volume>& p, Price price) { return p.first < price; }),
    mAsksCompare([](const std::pair<Price, Volume>& p, Price price) { return p.first > price; })
{}

void AddOrder(Side side, Price price, Volume volume)
{
    if (side == Side::Bid)
    {
        return AddOrder(mBidLevels, price, volume, mBidsCompare);
    }
    else
    {
        return AddOrder(mAskLevels, price, volume, mAsksCompare);
    }
}
```

Transport: networking & concurrency

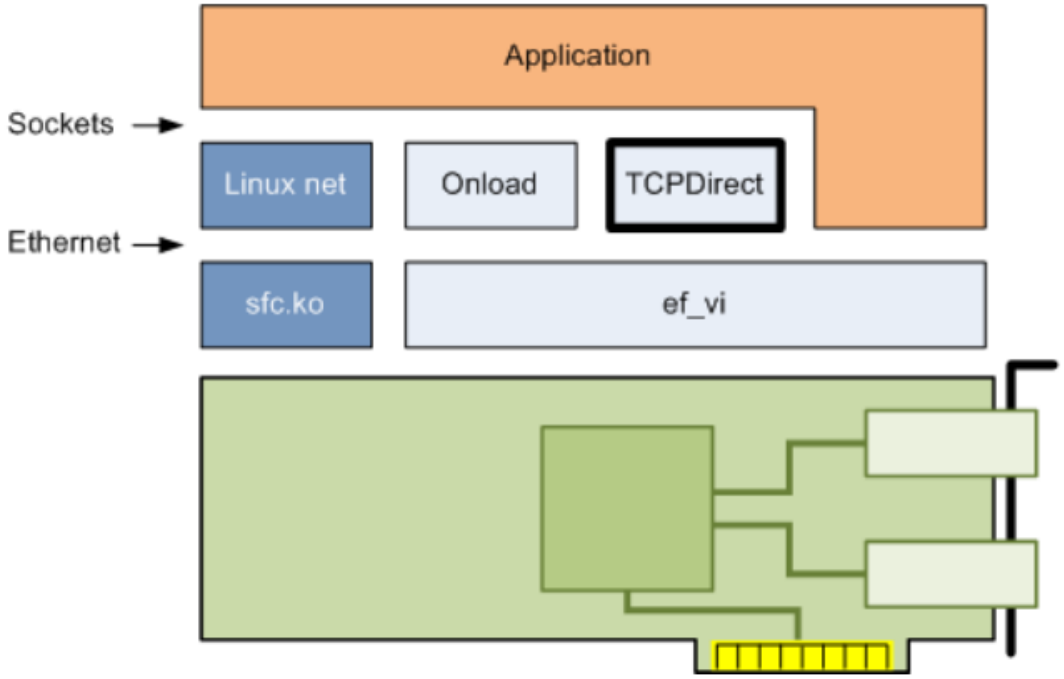


Low-latency transport

- General pattern
  - Kernel bypass when receiving data from the exchange (or other low-latency signals)
  - Dispatch / fan-out to processes on the same server

Userspace Networking

- Solarflare – industry standard low-latency NIC

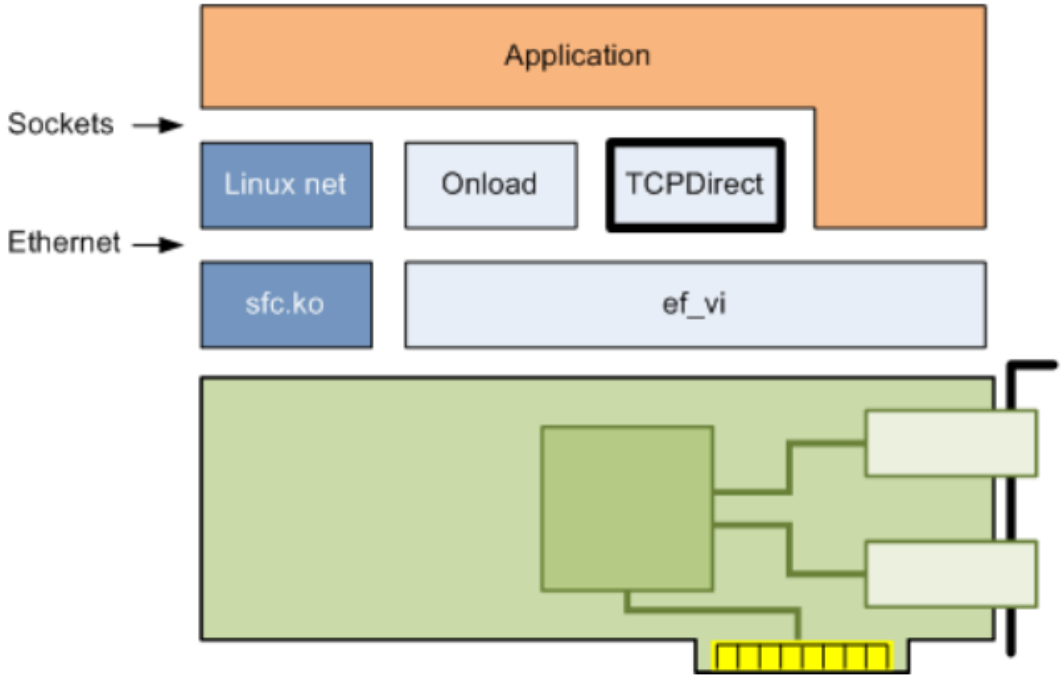


Source: solarflare.com

Userspace Networking

- Solarflare – industry standard low-latency NIC
- OpenOnload
  - Easy to use: compatible with BSD sockets (no code changes needed!)

```
onload --profile=latency ./algo_trader
```



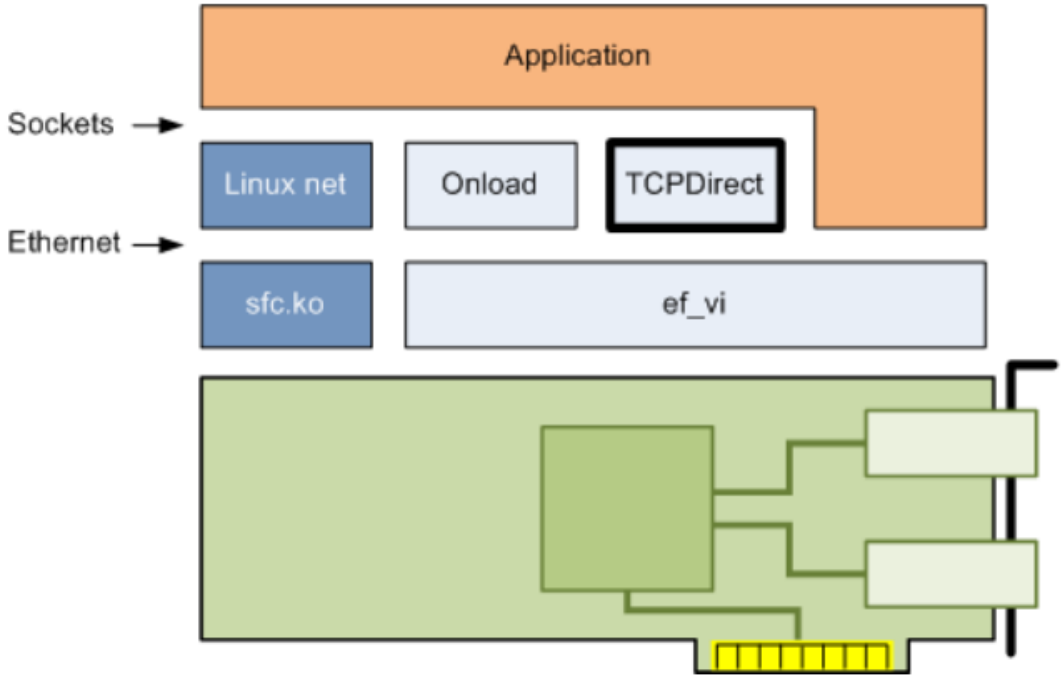
Source: solarflare.com

Userspace Networking

- Solarflare – industry standard low-latency NIC
- OpenOnload
  - Easy to use: compatible with BSD sockets (no code changes needed!)

```
onload --profile=latency ./algo_trader
```

- TCPDirect
  - Custom TCP/UDP stack
  - Reduced set of features



Source: solarflare.com

Userspace Networking

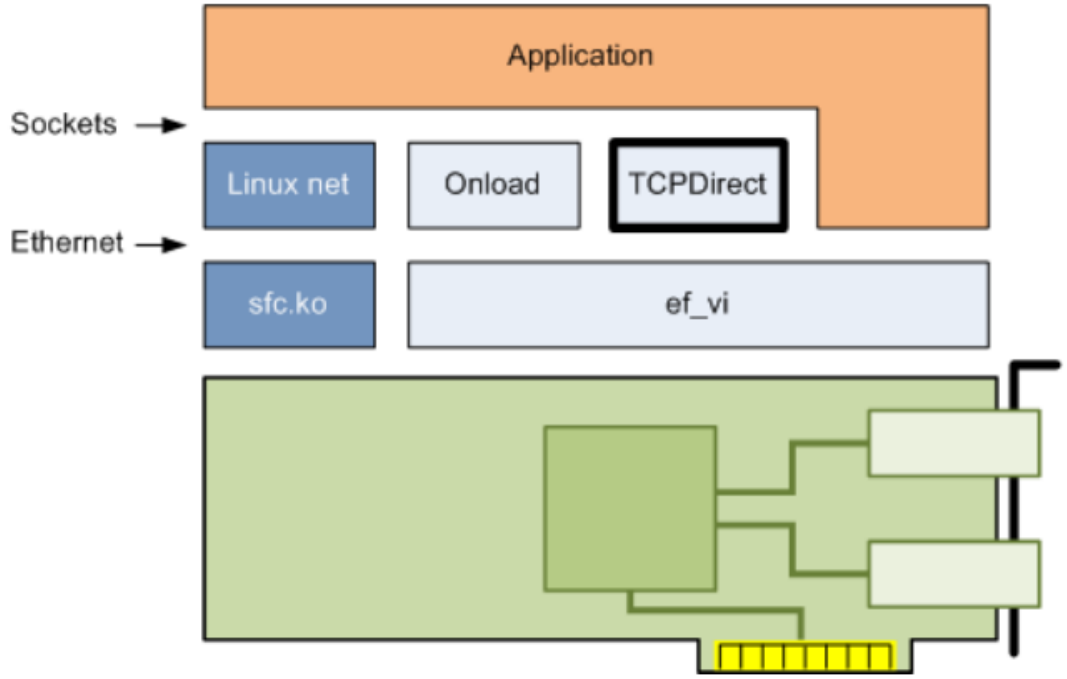
- Solarflare – industry standard low-latency NIC

- OpenOnload
  - Easy to use: compatible with BSD sockets (no code changes needed!)

```
onload --profile=latency ./algo_trader
```

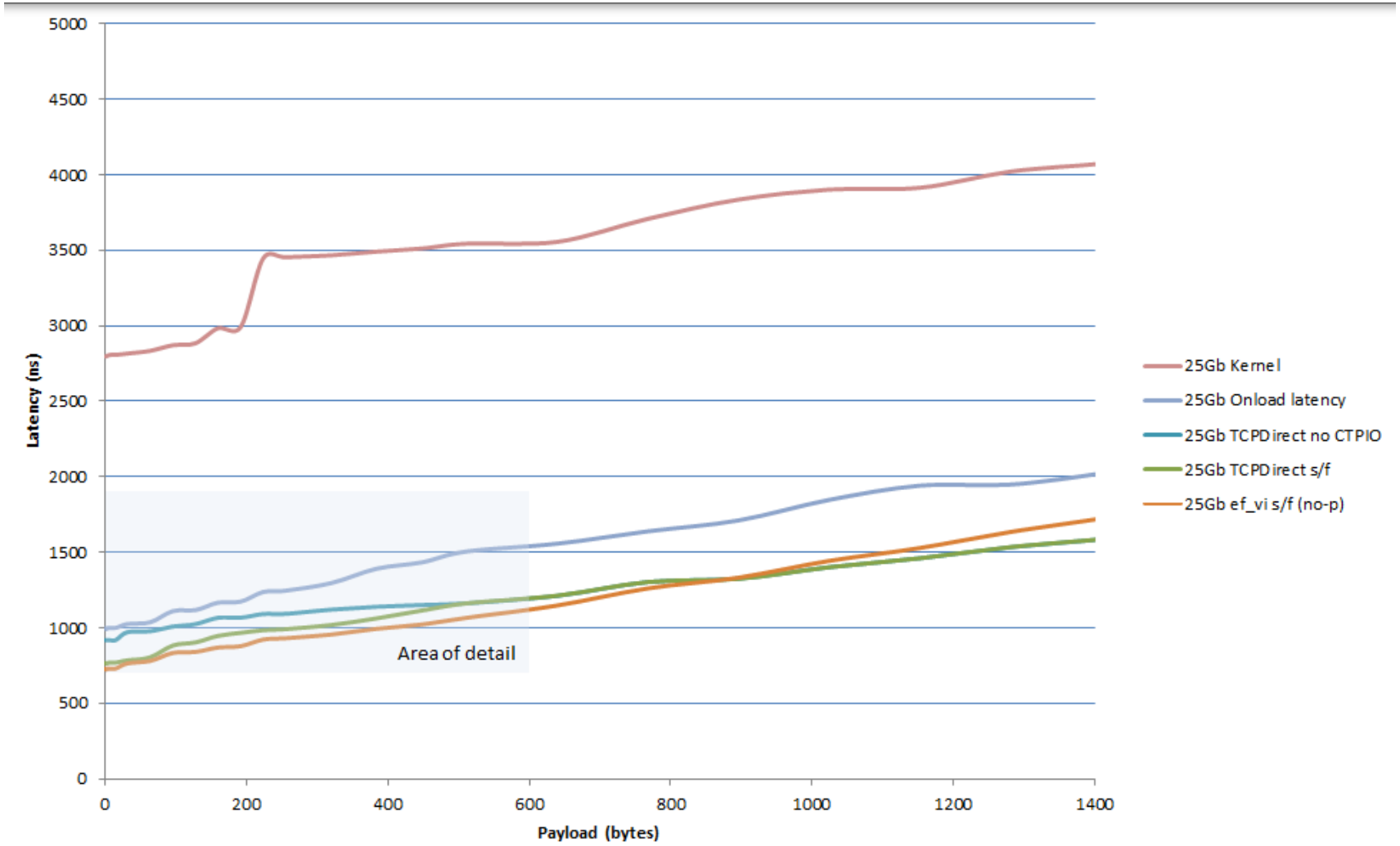
- TCPDirect
  - Custom TCP/UDP stack
  - Reduced set of features

- EF_VI
  - Layer 2 API: interface, buffers and memory management (similar to DPDK)
  - Lowest latency



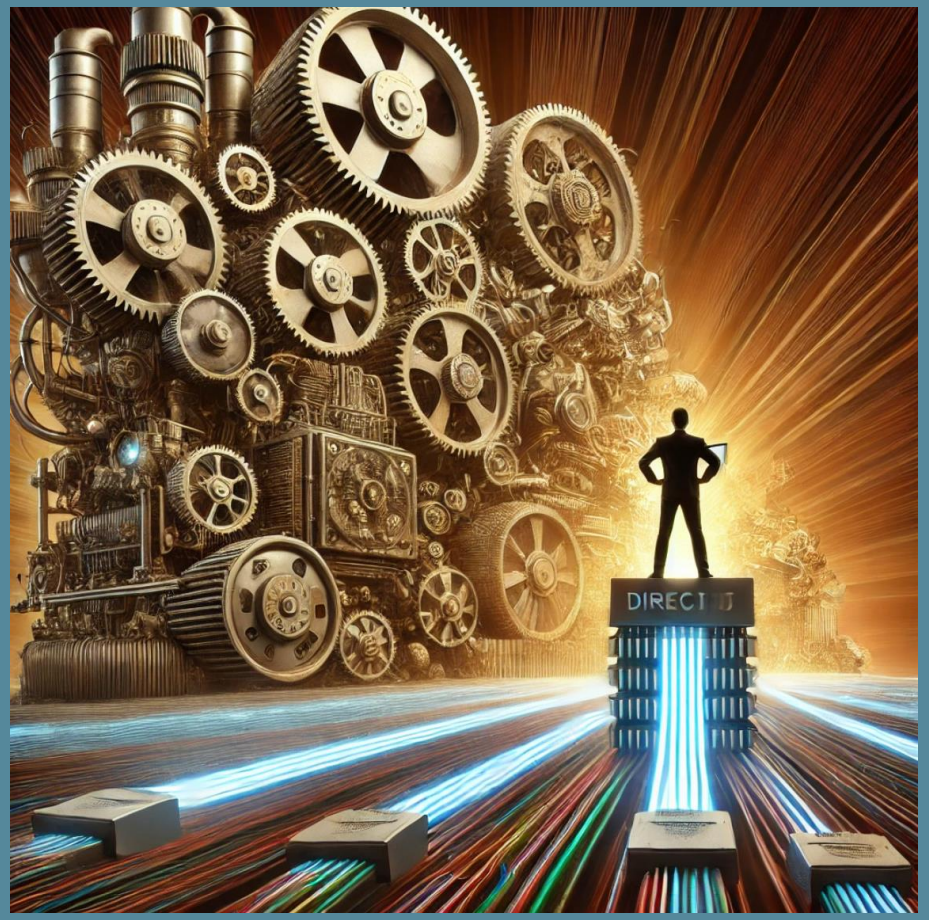
Source: solarflare.com

Userspace Networking



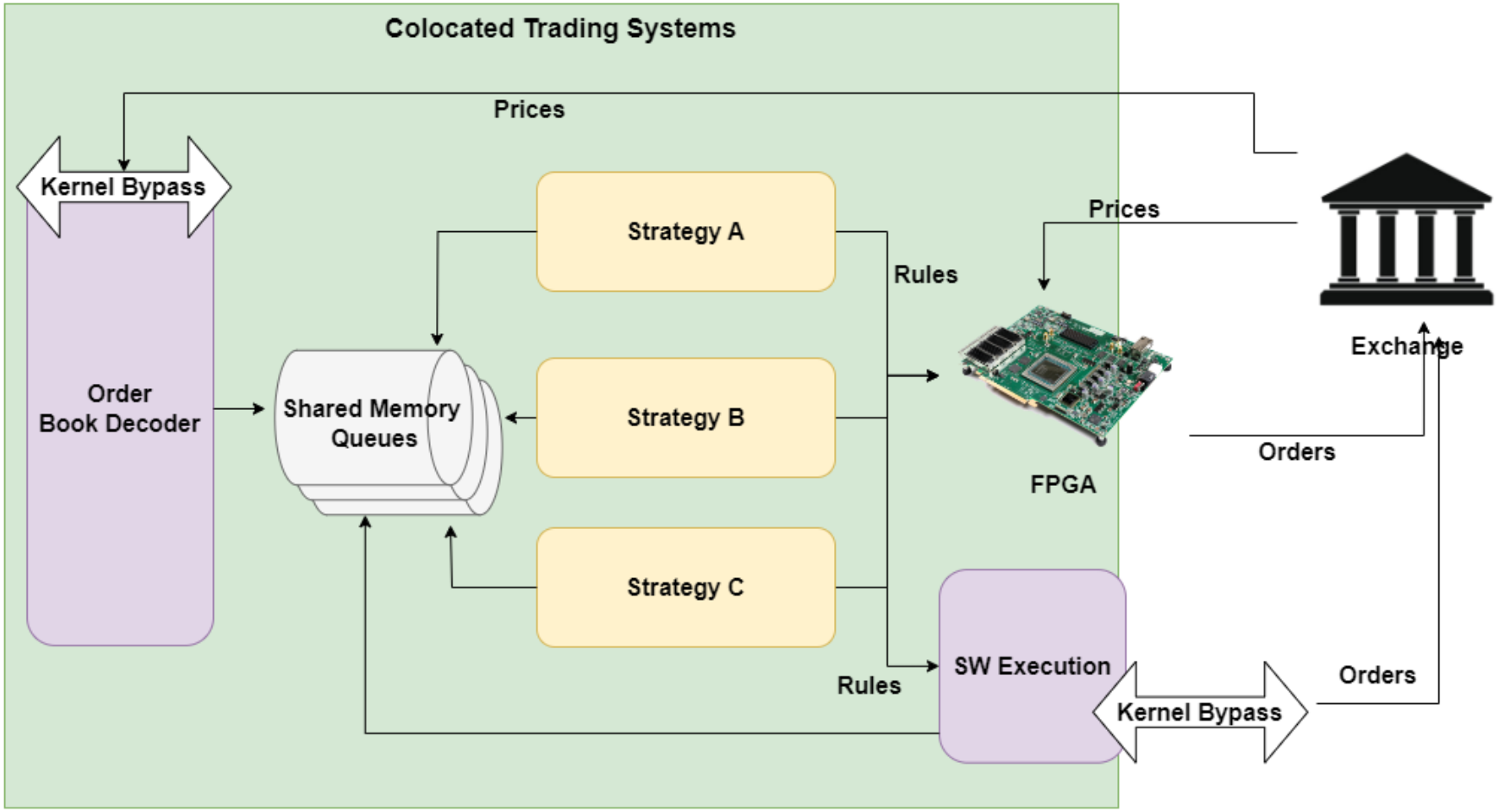
Source: docs.amd.com

Principle #6: “True efficiency is found not in the layers of complexity we add, but in the unnecessary layers we remove”



Source: ChatGPT

Connecting the dots



Shared Memory

- Why shared memory
  - If you don’t need sockets, no need to pay for their complexity
  - “As fast as it gets”
  - Kernel isn’t involved in any operations
  - Multi processes requires it – which is good for minimizing operational risk

Shared Memory

- Why shared memory
  - If you don’t need sockets, no need to pay for their complexity
  - “As fast as it gets”
  - Kernel isn’t involved in any operations
  - Multi processes requires it – which is good for minimizing operational risk

- What works well in shared memory
  - Contiguous blocks of data: arrays!
  - One writer, one or multiple readers → stay away from multiple writers!

Shared Memory

shm_open, mmap, munmap, shm_unlink, ftruncate, flock...

```
struct ProtocolHeader
{
    std::array<char, PROTOCOL_NAME_MAX_LENGTH> protocol_name;
    uint64_t magic_number;
    uint64_t buffer_size;
    uint32_t major_version;
    uint32_t minor_version;
    std::array<uint32_t, MAX_NUM_QUEUES> queue_size_bytes;
    uint8_t reserved[...];  // space for future protocol changes
} __attribute__((aligned));
```

Concurrent Queues

Bounded?	
Blocking?	
# Consumers?	
Message Size?	
Dispatch?	
Type Support?	

Concurrent Queues

Bounded?	Yes – simpler & faster
Blocking?	
# Consumers?	
Message Size?	
Dispatch?	
Type Support?	

Concurrent Queues

Bounded?	Yes – simpler & faster
Blocking?	No – readers don’t affect the writer
# Consumers?	
Message Size?	
Dispatch?	
Type Support?	

Concurrent Queues

Bounded?	Yes – simpler & faster
Blocking?	No – readers don’t affect the writer
# Consumers?	Many
Message Size?	
Dispatch?	
Type Support?	

Concurrent Queues

Bounded?	Yes – simpler & faster
Blocking?	No – readers don’t affect the writer
# Consumers?	Many
Message Size?	Variable length
Dispatch?	
Type Support?	

Concurrent Queues

Bounded?	Yes – simpler & faster
Blocking?	No – readers don’t affect the writer
# Consumers?	Many
Message Size?	Variable length
Dispatch?	Fan-out
Type Support?	

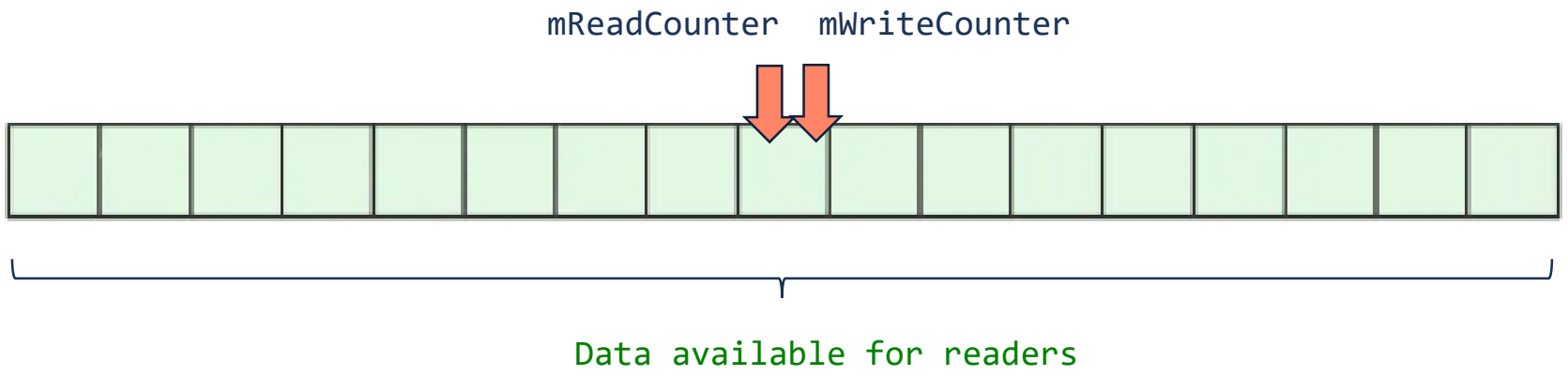
Concurrent Queues

Bounded?	Yes – simpler & faster
Blocking?	No – readers don’t affect the writer
# Consumers?	Many
Message Size?	Variable length
Dispatch?	Fan-out
Type Support?	PODs

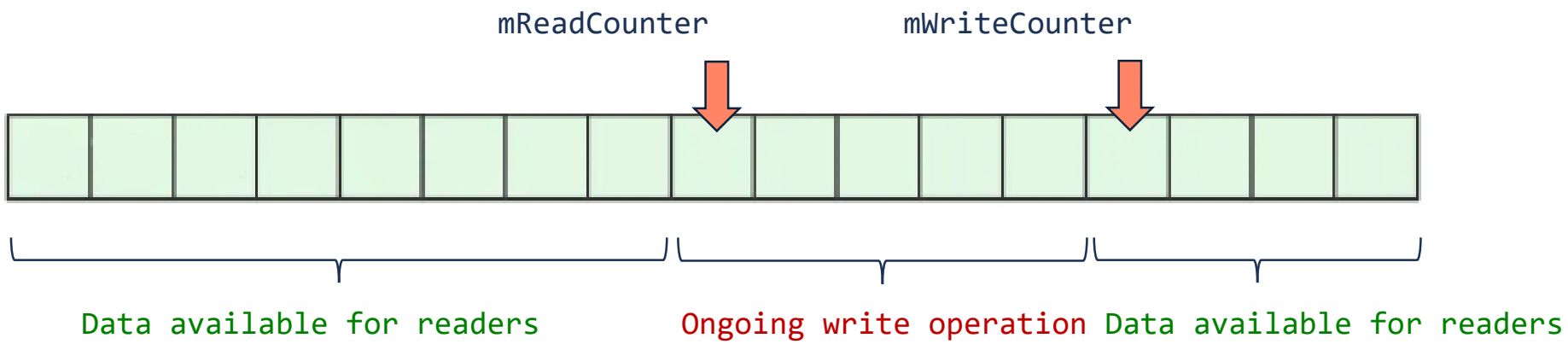
Principle #7: “Choose the right tool for the right task”

Source: ChatGPT

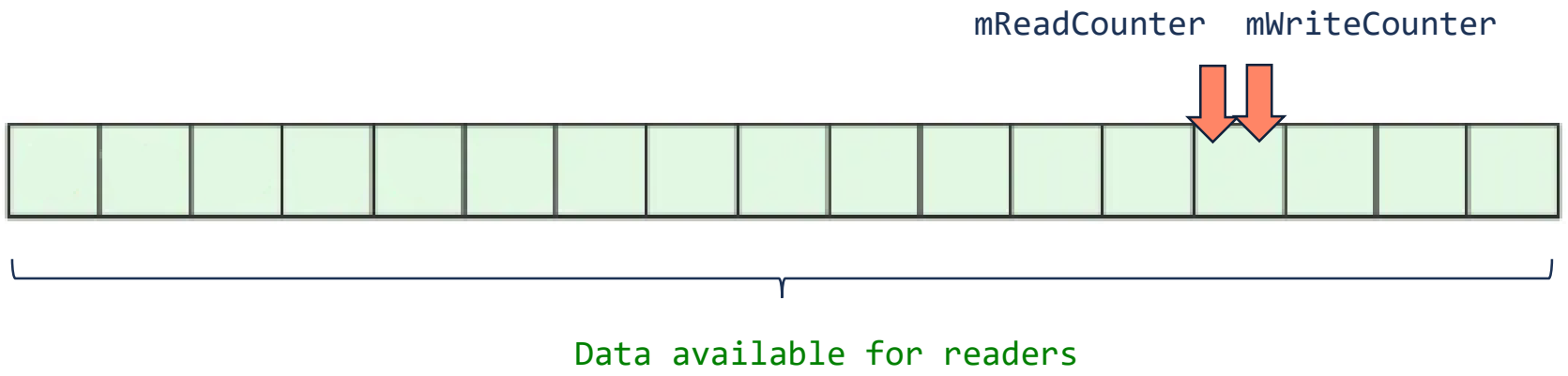
FastQueue – Design



FastQueue – Design

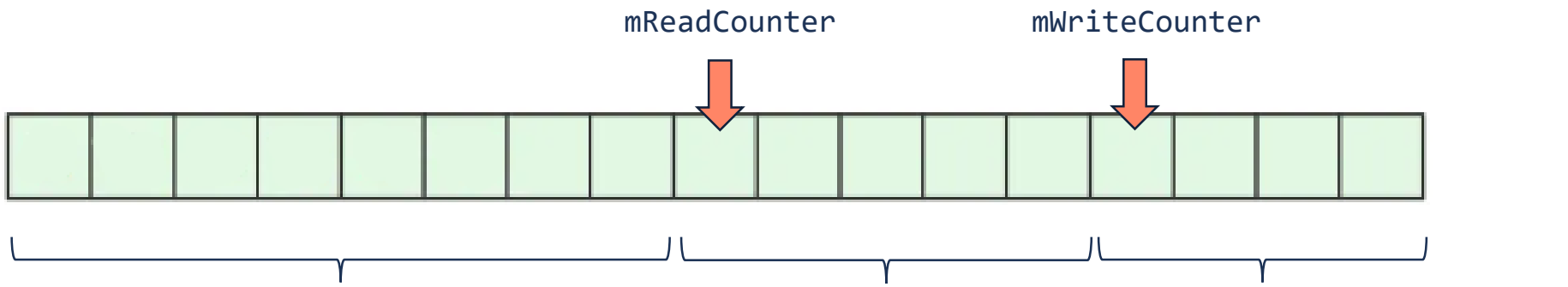


FastQueue – Design



FastQueue – Design

```
struct FastQueue
{
    // Both counters are written by the producer, read by the consumer(s).
    // Before/after a Write operation, both counters contain the same value.

    // [WriteCounter,ReadCounter] defines the area where data can be read
    alignas(CACHE_LINE_SIZE) std::atomic<uint64_t> mReadCounter{0};

    // [ReadCounter,WriteCounter] defines the area where data is being written to
    alignas(CACHE_LINE_SIZE) std::atomic<uint64_t> mWriteCounter{0};

    alignas(CACHE_LINE_SIZE) uint8_t mBuffer[0];
};
```



FastQueue – API

```
struct QProducer
{
    void Write(std::span<std::byte> buffer);
};


struct QConsumer
{
    int32_t TryRead(std::span<std::byte> buffer); // returns #bytes read, 0 if nothing to read
};
```

FastQueue – Writer

```
// simplified code!

void QProducer::Write(std::span<std::byte> buffer)
{
    const int32_t payloadSize = sizeof(int32_t) + buffer.size();
    mLocalCounter += payloadSize;

    mQ->mWriteCounter.store(mLocalCounter, std::memory_order_release);

    std::memcpy(mNextElement, &size, sizeof(int32_t));
    std::memcpy(mNextElement + sizeof(int32_t), buffer.data(), buffer.size());

    mQ->mReadCounter.store(mLocalCounter, std::memory_order_release);

    mNextElement += payloadSize;
}
```

FastQueue – Reader

```
int32_t QConsumer::TryRead(std::span<std::byte> buffer)
{
    if (mLocalCounter == mQ->mReadCounter.load(std::memory_order_acquire))
        return 0;

    int32_t size;
    std::memcpy(&size, mNextElement, sizeof(int32_t));

    int32_t writeCounter = mQ->mWriteCounter.load(std::memory_order_acquire);
    EXPECT(writeCounter – mLocalCounter <= QUEUE_SIZE, "queue overflow");
    EXPECT(size <= buffer.size(), "buffer space isn’t large enough");

    std::memcpy(buffer.data(), mNextElement + sizeof(size), size);

    const int32_t payloadSize = sizeof(size) + size;
    mLocalCounter += payloadSize;
    mNextElement += payloadSize;

    writeCounter = mQ->mWriteCounter.load(std::memory_order_acquire);
    EXPECT(writeCounter – mLocalCounter <= QUEUE_SIZE, "queue overflow");
}
```

FastQueue – Reader

```
int32_t QConsumer::TryRead(std::span<std::byte> buffer)
{
    if (mLocalCounter == mQ->mReadCounter.load(std::memory_order_acquire))
        return 0;

    int32_t size;
    std::memcpy(&size, mNextElement, sizeof(int32_t);

    int32_t writeCounter = mQ->mWriteCounter.load(std::memory_order_acquire);
    EXPECT(writeCounter – mLocalCounter <= QUEUE_SIZE, "queue overflow");
    EXPECT(size <= buffer.size(), "buffer space isn't large enough");

    std::memcpy(buffer.data(), mNextElement + sizeof(size), size);

    const int32_t payloadSize = sizeof(size) + size;
    mLocalCounter += payloadSize;
    mNextElement += payloadSize;

    writeCounter = mQ->mWriteCounter.load(std::memory_order_acquire);
    EXPECT(writeCounter – mLocalCounter <= QUEUE_SIZE, "queue overflow");
}
```

Data race! (6.9.2.2)

P1478R5: Byte-wise atomic memcpy

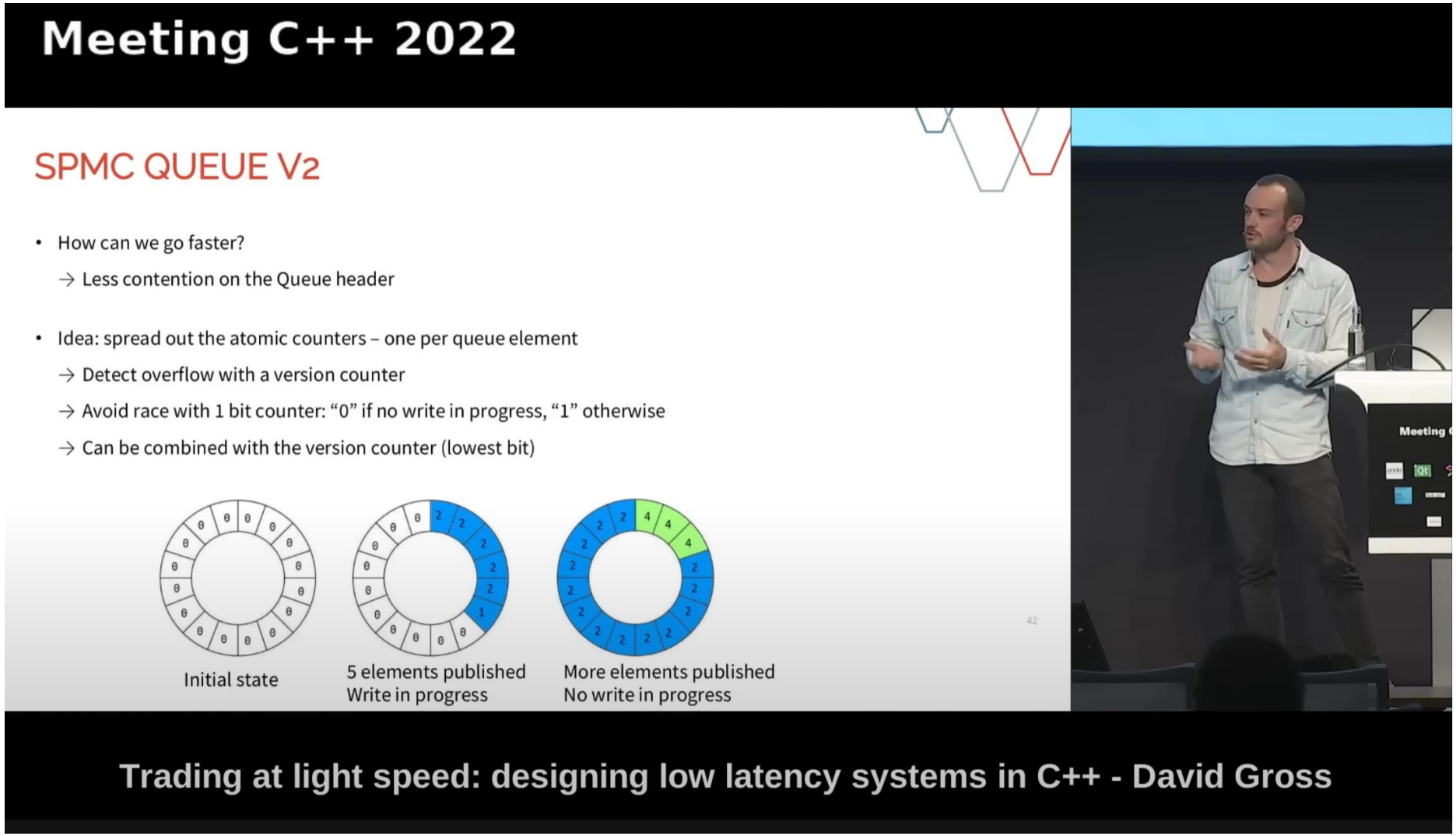
FastQueue – Performance Measurements

- AMD EPYC 9474F
  - Tuned for low-latency, core isolated, etc
- Each message is 73 bytes
- Queue is 8 MB

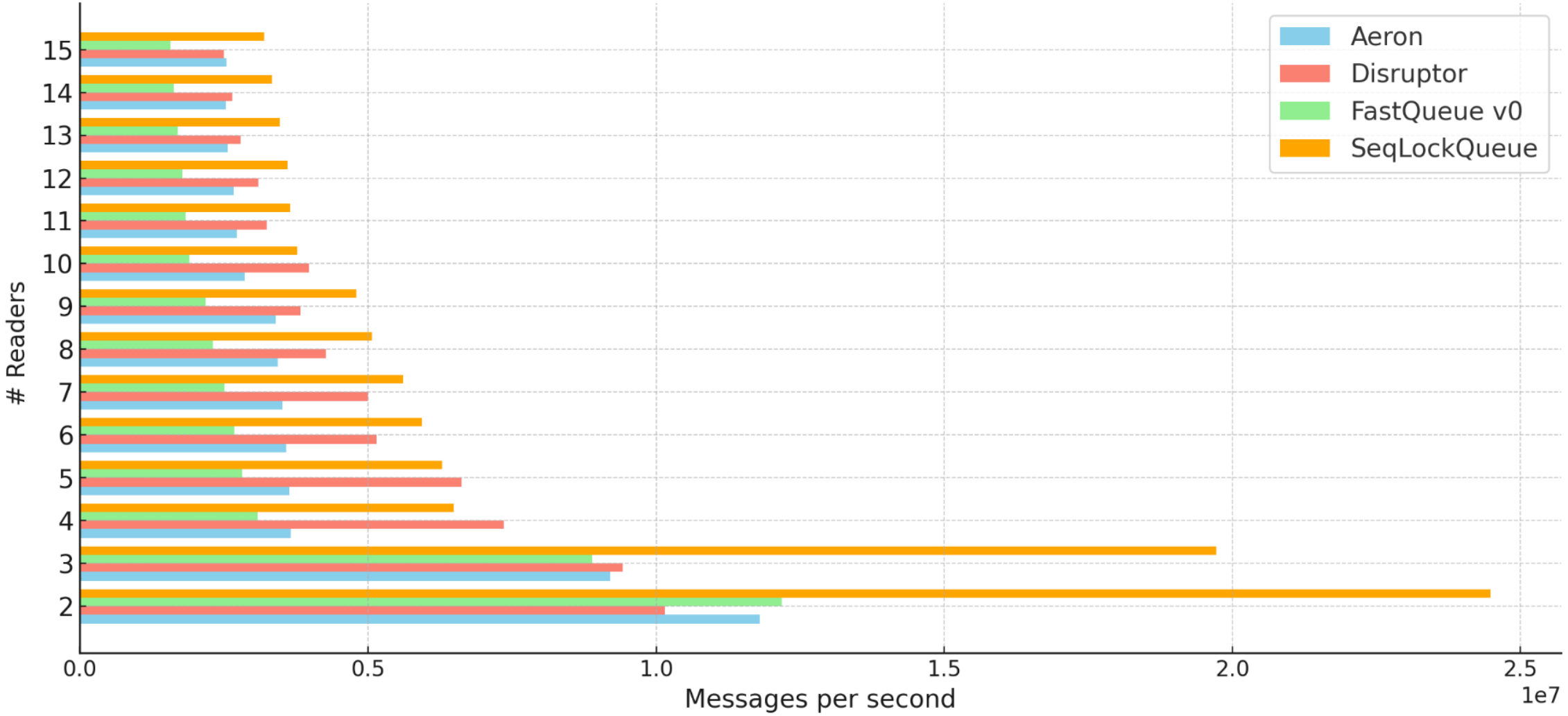
FastQueue – Performance Measurements

- AMD EPYC 9474F
  - Tuned for low-latency, core isolated, etc
- Each message is 73 bytes
- Queue is 8 MB
- Compared against
  - Aeron: C++ client only - concurrent/broadcast/Broadcast{Transmitter,Receiver.h}
  - Disruptor (Java): com.lmax.disruptor.Ringbuffer
- SPMC queues but unicast queues – therefore left out
  - Folly MPMC
  - moodycamel::ConcurrentQueue

SeqLockQueue



FastQueue – Baseline



Optimization #1: Caching the Write Counter

```
void QProducer::Write(std::span<std::byte> buffer)
{
    const int32_t payloadSize = sizeof(int32_t) + buffer.size();
    mLocalCounter += payloadSize;

    // we "reserve" more space (X% of the total queue)
    // to avoid touching this cache line on every message written
    if (mCachedWriteCounter < mLocalCounter)
    {
        mCachedWriteCounter = Align<Q_WRITE_COUNTER_BLOCK_BYTES>(mLocalCounter);
        mQ->mWriterCounter.store(mCachedWriteCounter, std::memory_order_release);
    }

    std::memcpy(mNextElement, &size, sizeof(int32_t));
    std::memcpy(mNextElement + sizeof(int32_t), buffer.data(), buffer.size());

    mQ->mReadCounter.store(mLocalCounter, std::memory_order_release);

    mNextElement += payloadSize;
}
```

Optimization #1: Caching the Write Counter

```
void QProducer::Write(std::span<std::byte> buffer)
{
    const int32_t payloadSize = sizeof(int32_t) + buffer.size();
    mLocalCounter += payloadSize;

    // we "reserve" more space (X% of the total queue)
    // to avoid touching this cache line on every message written
    if (mCachedWriteCounter < mLocalCounter)
    {
        mCachedWriteCounter = Align<Q_WRITE_COUNTER_BLOCK_BYTES>(mLocalCounter);
        mQ->mWriterCounter.store(mCachedWriteCounter, std::memory_order_release);
    }

    std::memcpy(mNextElement, &size, sizeof(int32_t));
    std::memcpy(mNextElement + sizeof(int32_t), buffer.data(), buffer.size());

    mQ->mReadCounter.store(mLocalCounter, std::memory_order_release);

    mNextElement += payloadSize;
}
```

```
template <size_t S, class T>
inline T Align(T value)
{
    return (value + (S - 1)) & (~(S - 1));
}
```

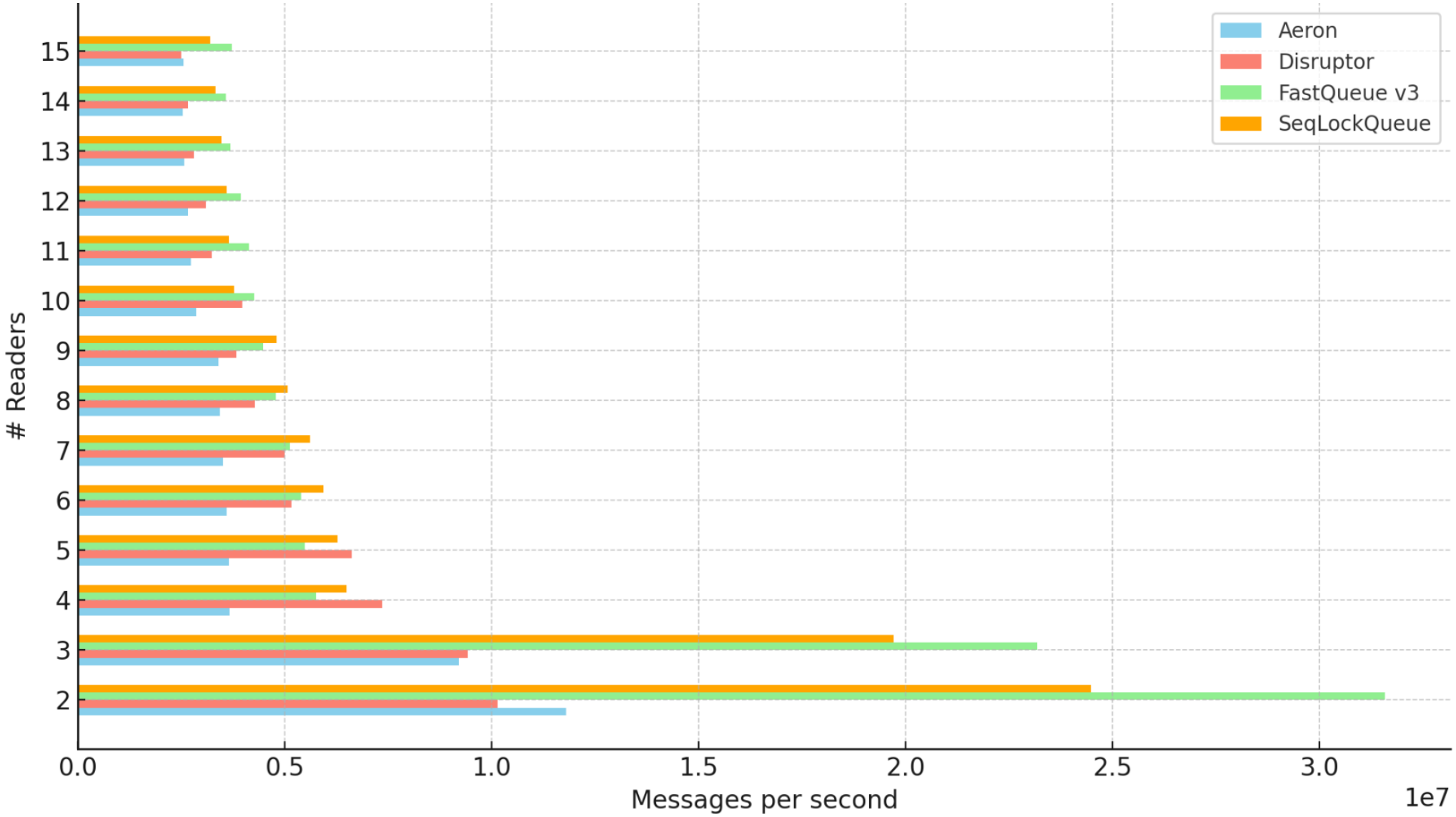
Optimization #2: Data Alignment

```
void QProducer::Write(std::span<std::byte> buffer)
{
    const int32_t payloadSize = sizeof(int32_t) + Align<Q_BLOCK_ALIGNMENT>(buffer.size());
    mLocalCounter += payloadSize;

    ...
}
```

Optimization #3: Caching the Read Counter

```
int32_t QConsumer::TryRead(std::span<std::byte> buffer)
{
    // we might already know from the previous read counter that more data is available, and
    // in this case we avoid reading this cache line for no reason
    if (mLocalReadCounter == mCachedReadCounter)
    {
        mCachedReadCounter = mQ->mReadCounter.load(std::memory_order_acquire);
    }

    if (mLocalReadCounter == mCachedReadCounter)
    {
        return 0;
    }

    ...
}
```

FastQueue – Final Results on AMD EPYC 9474F



Principle #4: “Simplicity is the ultimate sophistication”

Source: ChatGPT

Going further – Zero copy

```
struct QProducer
{
    void Write(std::span<std::byte> buffer);
};
```

Going further – Zero copy

```
struct QProducer
{
    void Write(std::span<std::byte> buffer);
};
```

Better API: you can serialize directly into the queue, without an additional copy!

```
template <class C>
void Write(int32_t size, C c)
{
    std::span<std::byte> buffer = GetBuffer(size);
    c(buffer);
    Flush();
}
```

Going further – Zero copy

```
struct QProducer
{
    void Write(std::span<std::byte> buffer);
};
```

Better API: you can serialize directly into the queue, without an additional copy!

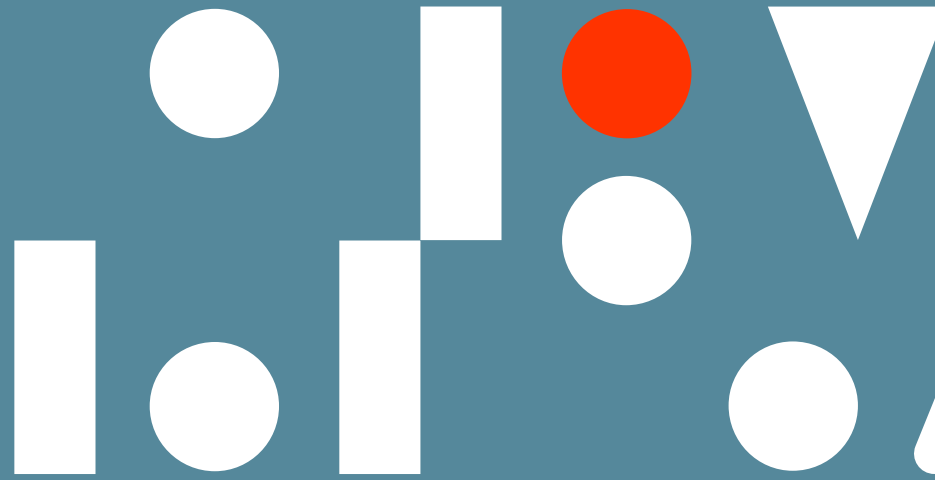
```
template <class C>
void Write(int32_t size, C c)
{
    std::span<std::byte> buffer = GetBuffer(size);
    c(buffer);
    Flush();
}
```

- Common serialization libraries used
  - Simple Binary Encoding (SBE)
  - CapnProto
  - FlatBuffers
  - Protobuf

FastQueue – Going even further

- Bulk Writing
- Duplicate Queue Header on “remote” NUMA node

Measurements in low-latency trading systems



Measurements in low-latency trading systems

```
void Executor::PollMarketData()
{
    while (poll([&](const auto& msg)
    {
        if (IsInteresting(msg))
        {
            SendOrder();
        }
    }));
}
```

Measurements in low-latency trading systems

```
void Executor::SendOrder()
{
    ScopedTrace trace(__FUNCTION__, ...);
    ...
}


struct ScopedTrace
{
    ScopedTrace(...) :
        mStartTs(__rdtsc())
    {}

    ~ScopedTrace()
    {
        Write(__rdtsc() - mStartTs);
    }
};
```

Measurements in low-latency trading systems

```
void Executor::SendOrder()
{
    ScopedTrace trace(__FUNCTION__, ...);
    ...
}
```

```
struct ScopedTrace
{
    ScopedTrace(...) :
        mStartTs(__rdtsc())
    {}

    ~ScopedTrace()
    {
        Write(__rdtsc() - mStartTs, ...);
    }
};
```

1. Write TSC interval + metadata to a queue
2. Separate thread writing data to disk
   (or publishing it live, or both)

“Finding the next bottleneck is like chasing shadows in a dark jungle—always elusive, until you're already tangled in its grasp”

Source: ChatGPT

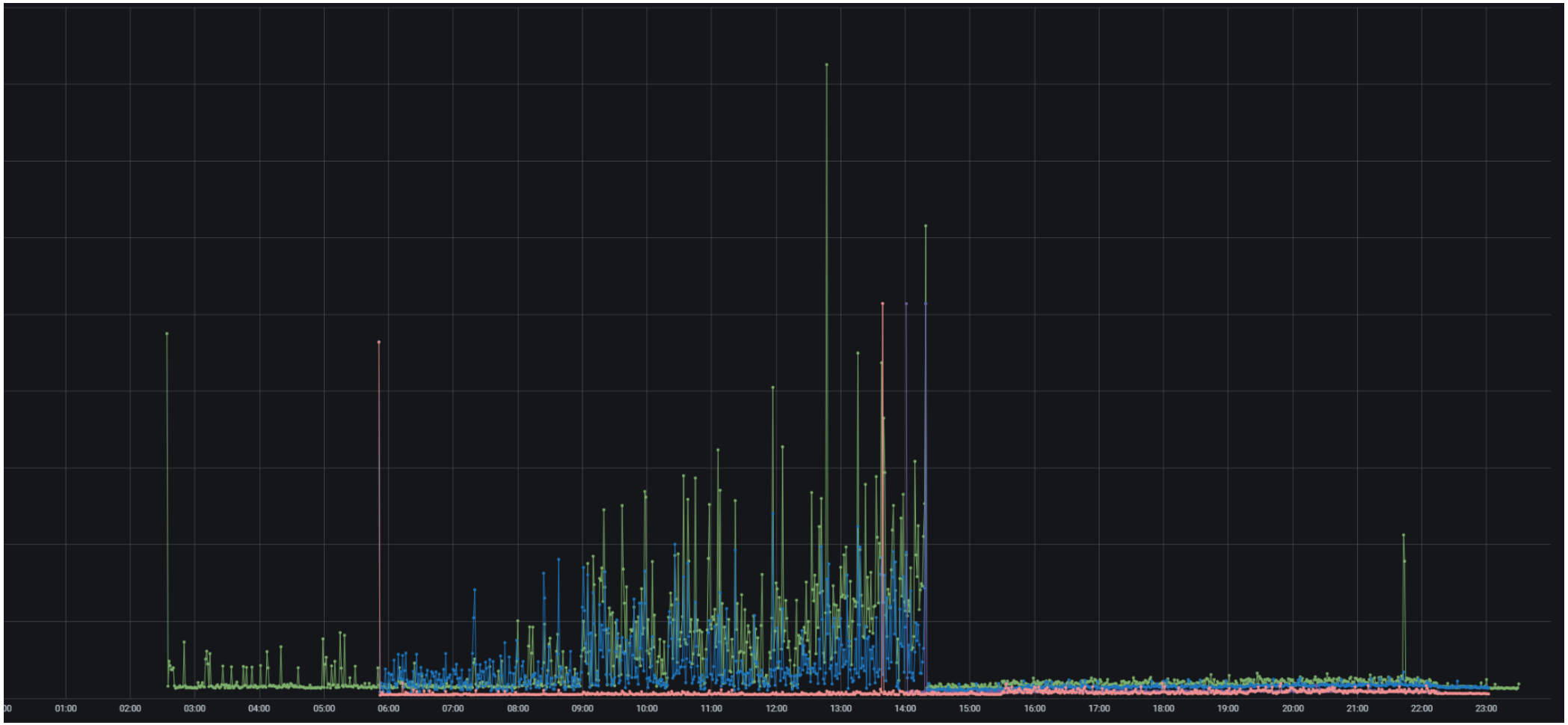
Clang Xray Instrumentation

```
__xray_patch_function(id);

__xray_set_handler(XRayRCon::Profile);
```

Clang Xray Instrumentation

```
__xray_patch_function(id);

__xray_set_handler(XRayRCon::Profile);


[[clang::xray_never_instrument]] void XRayRCon::Profile(int32_t index, XRayEntryType entryType)
{
    if (entryType == ENTRY)
    {
        Optiver::Profiling::StartTrace(index);
    }
    else if (entryType == EXIT || entryType == TAIL)
    {
        Optiver::Profiling::StopTrace();
    }
}
```

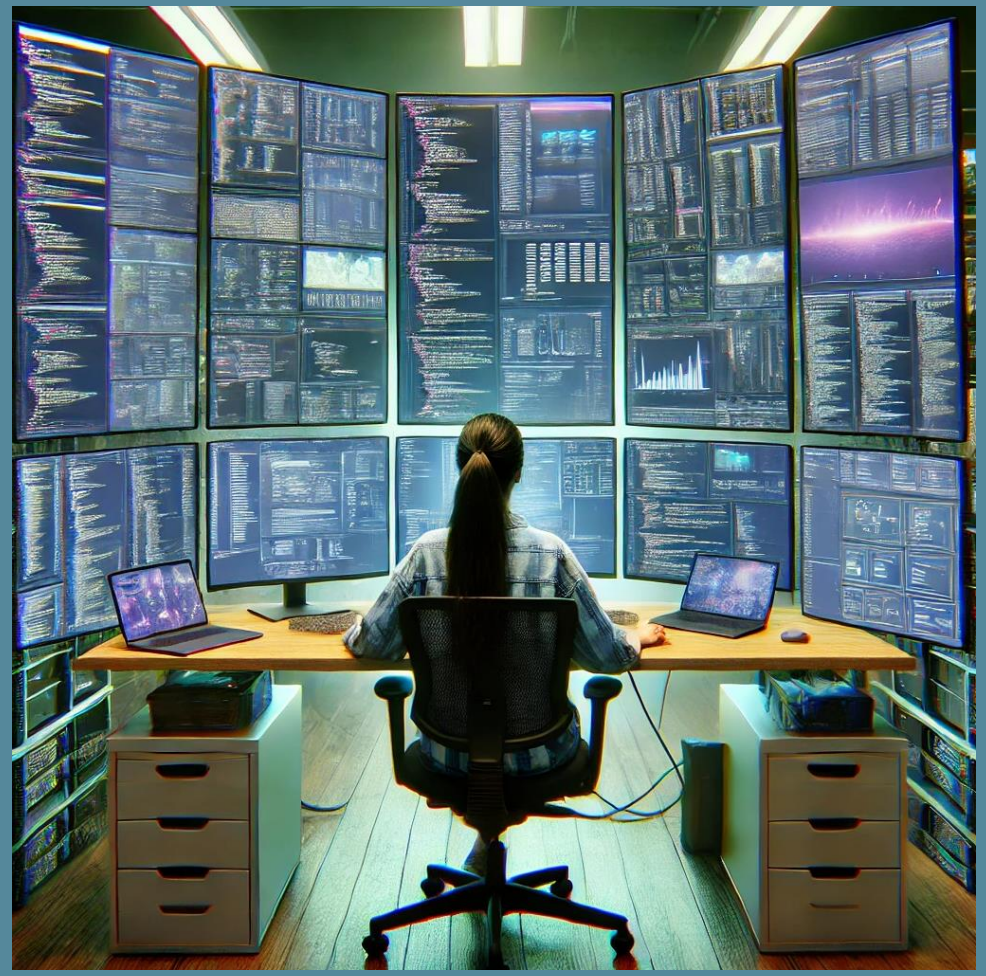
Metrics, audits and trends



Principle #8: “Being fast is good – staying fast is better”

Source: ChatGPT

Outro – You’re not alone



Source: ChatGPT

You’re not alone

```
const size_t kbytes = std::atoi(argv[1]);
const size_t n = kbytes * 1024 / sizeof(uint64_t);

const std::vector<uint64_t> v = GenerateShuffledIndices(n);


for (auto& state : _)
{
    const uint64_t sum = Sum(v, n);
    benchmark::DoNotOptimize(sum);
}


state.SetBytesProcessed(n * sizeof(uint64_t) * state.iterations() * state.range(0));
```

You’re not alone

```
const size_t kbytes = std::atoi(argv[1]);
const size_t n = kbytes * 1024 / sizeof(uint64_t);

const std::vector<uint64_t> v = GenerateShuffledIndices(n);
```

```
std::mt19937 gen(std::random_device{}());
std::vector<uint64_t> v(n);
std::iota(v.begin(), v.end(), 0);
std::shuffle(v.begin(), v.end(), gen);
```

```
for (auto& state : _)
{
    const uint64_t sum = Sum(v, n);
    benchmark::DoNotOptimize(sum);
}
```

```
state.SetBytesProcessed(n * sizeof(uint64_t) * state.iterations() * state.range(0));
```

You’re not alone

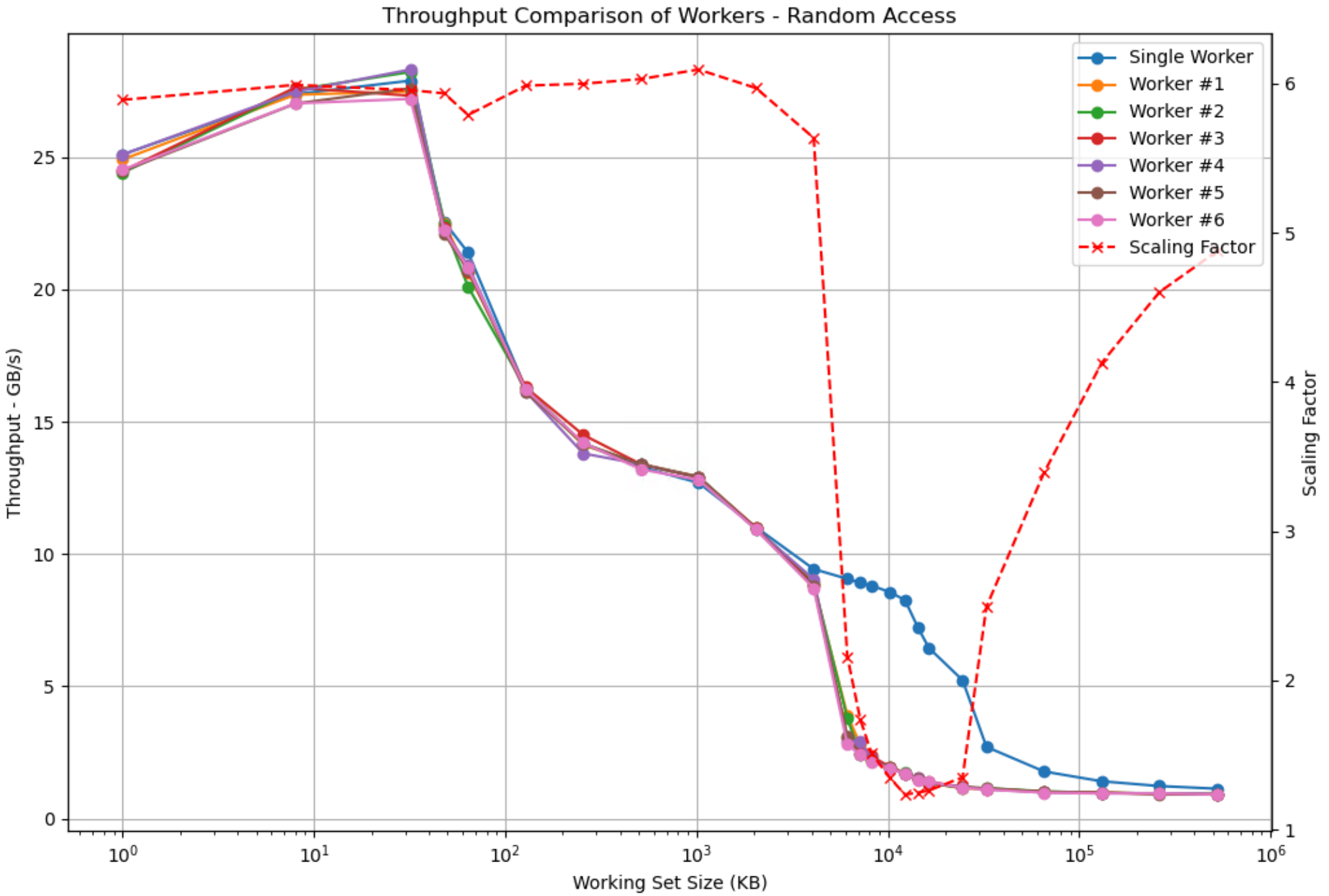
```
const size_t kbytes = std::atoi(argv[1]);
const size_t n = kbytes * 1024 / sizeof(uint64_t);

const std::vector<uint64_t> v = GenerateShuffledIndices(n);
```

```
std::mt19937 gen(std::random_device{}());
std::vector<uint64_t> v(n);
std::iota(v.begin(), v.end(), 0);
std::shuffle(v.begin(), v.end(), gen);
```

```
for (auto& state : _)
{
    const uint64_t sum = Sum(v, n);
    benchmark::DoNotOptimize(sum);
}
```

```
for (size_t pos = 0; pos < n; ++pos)
{
    sum += v[v[pos]];
}
```

```
state.SetBytesProcessed(n * sizeof(uint64_t) * state.iterations() * state.range(0));
```


Throughput Comparison of Workers - Random Access
Single Worker
Worker #1
Worker #2
Worker #3
Worker #4
Worker #5
Worker #6
Scaling Factor
Throughput - GB/s
Scaling Factor
Working Set Size (KB)
0
5
10
15
20
25
1
2
3
4
5
6
10^0
10^1
10^2
10^3
10^4
10^5
10^6

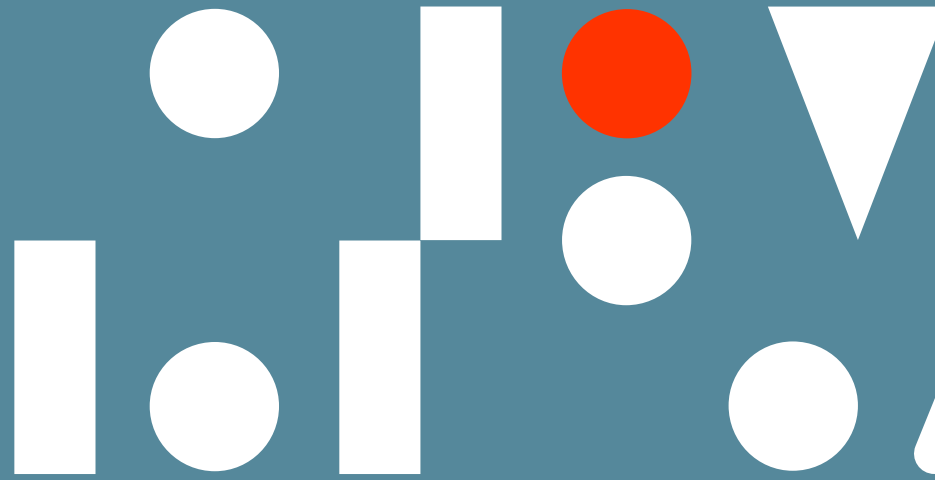
Principle #9: “Thinking about the system as a whole”

Source: ChatGPT

Principle #10: “The performance of your code depends on your colleagues’ code as much as yours”

Source: ChatGPT

Final thoughts



Final Thoughts

- Low-latency programming is also a losers' game!
  - It's mostly about being very disciplined…
  - … and not over-engineering your system
- In the trading game, time to market is an important "latency" too!

Credits

- Optiver UK – Alex Brady
- Optiver Chicago – Rishabh Thakkar
- Optiver Amsterdam – Yoram Versluis, Casey Williams
- Moncef Mechri
- Maciek Gajewski

Going further

- What Every Programmer Should Know About Memory Ulrich Drepper
- AMD X3 NIC Low Latency Quickstart
- Optimizing RHEL 9 for Real Time for low latency
- Can seqlocks get along with programming language memory models? Hans-J. Boehm
- Unlocking Modern CPU Power - Next-Gen C++ Optimization Techniques Fedor G Pikus
- Data-Oriented Design and C++ Mike Acton
- Trading at light speed: designing low latency systems in C++ David Gross

Optiver

Questions?