Qminers

Level Up the Book

(The Devil's in the Levels)

Mgr. Petr Filipský

Hi, my name is **Petr Filipský**, and I've been part of the **Qminers C++ team** for the past seven years.
Today's presentation is called **Level Up the Book**.
I've packed a lot of material into the slides, so let's dive right in.

Company

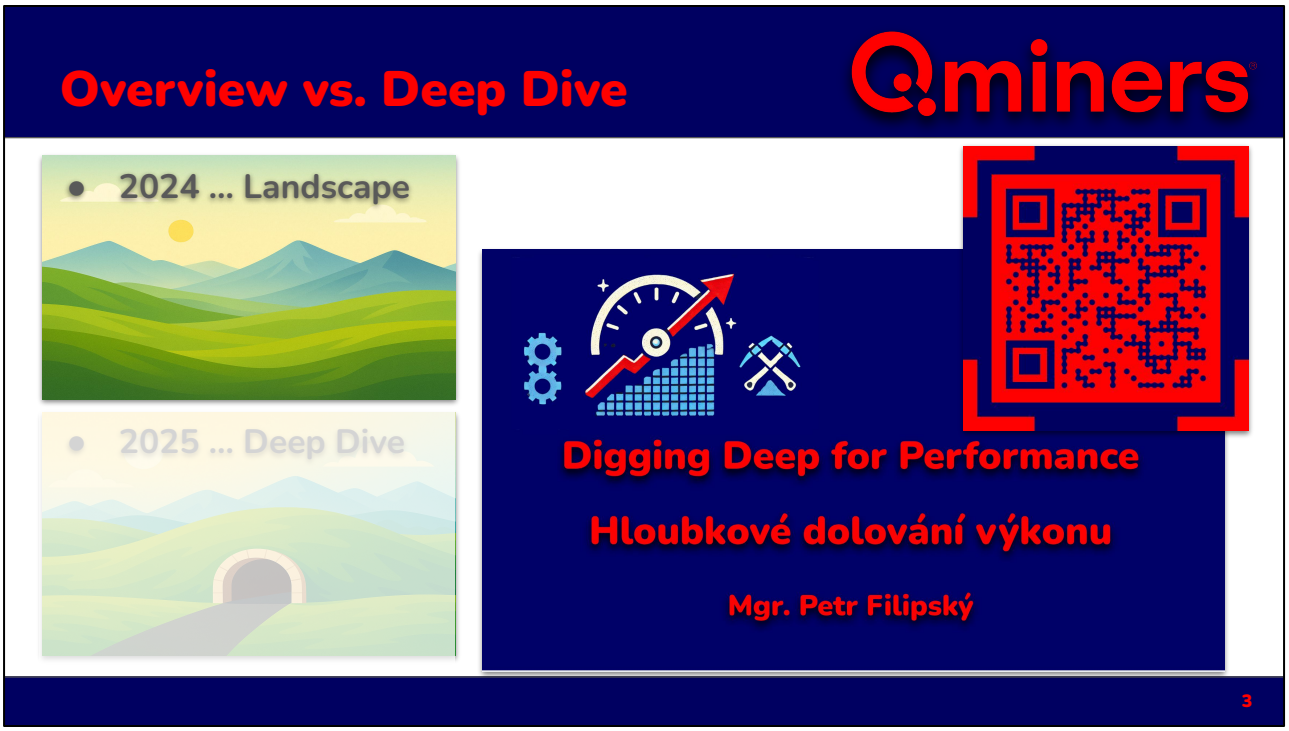
Qminers

- Who we are
- What we are doing
- How we do it

2

Let me start by saying a few words about Qminers - the company where I work.

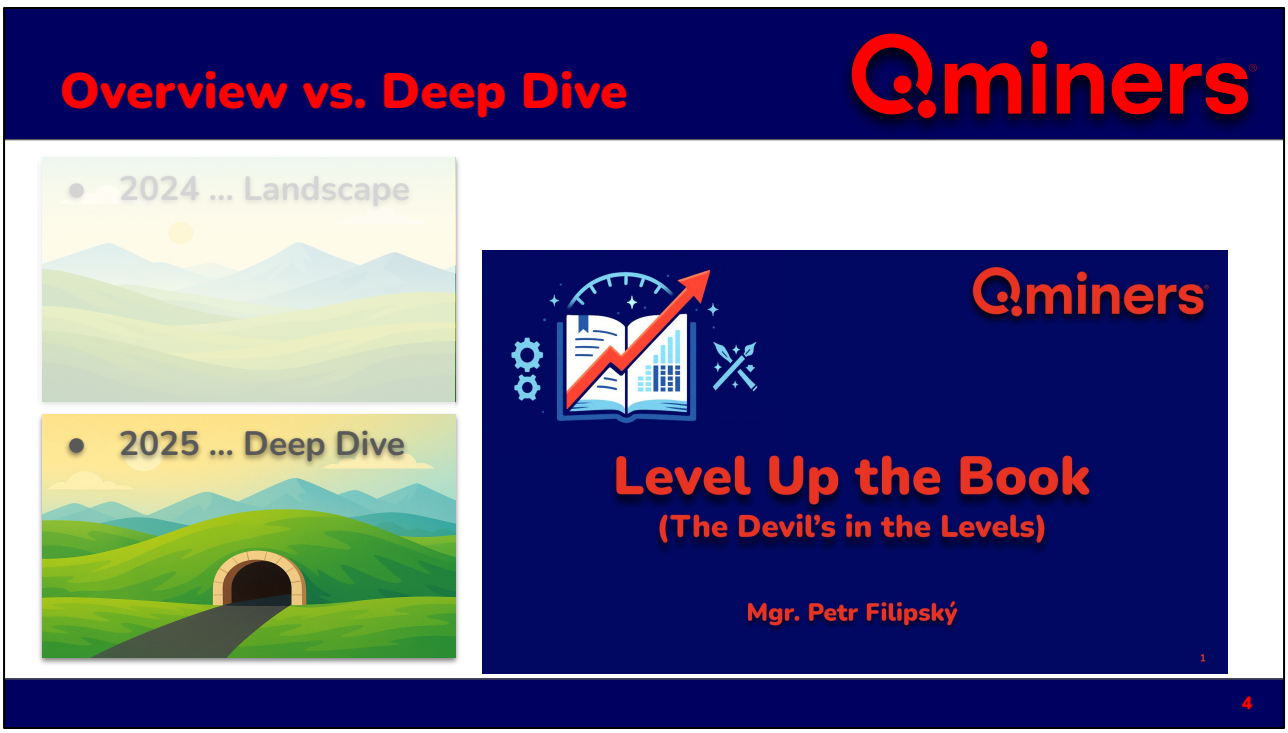
- **Who we are**
  Qminers was founded thirteen years ago, and we're based in the center of Prague, in the beautiful Špork Palace on Hybernská Street.
- **What we do**
  We specialize in high-frequency trading - that is, algorithmic trading on global financial markets.
  Our software trades fully autonomously on exchanges around the world, using mathematical models developed by our in-house analysts.
- **What we use**
  We develop primarily in C++ - there are six of us on the C++ team, and we're currently looking to hire at least two more engineers.
  We also use Python - roughly the same number of developers - for data visualization, reporting tools, and analytical infrastructure that supports our team of about 25 analysts.



This isn’t my first time here at Matfyz - in fact, I studied here many years ago.
And last year, I also gave a talk here on St. Nicholas Day.

The talk was called *Digging Deep for Performance*, and it was more of a general overview - a broad map of software optimization techniques.
So if you're interested and haven’t seen it yet, here’s the link.

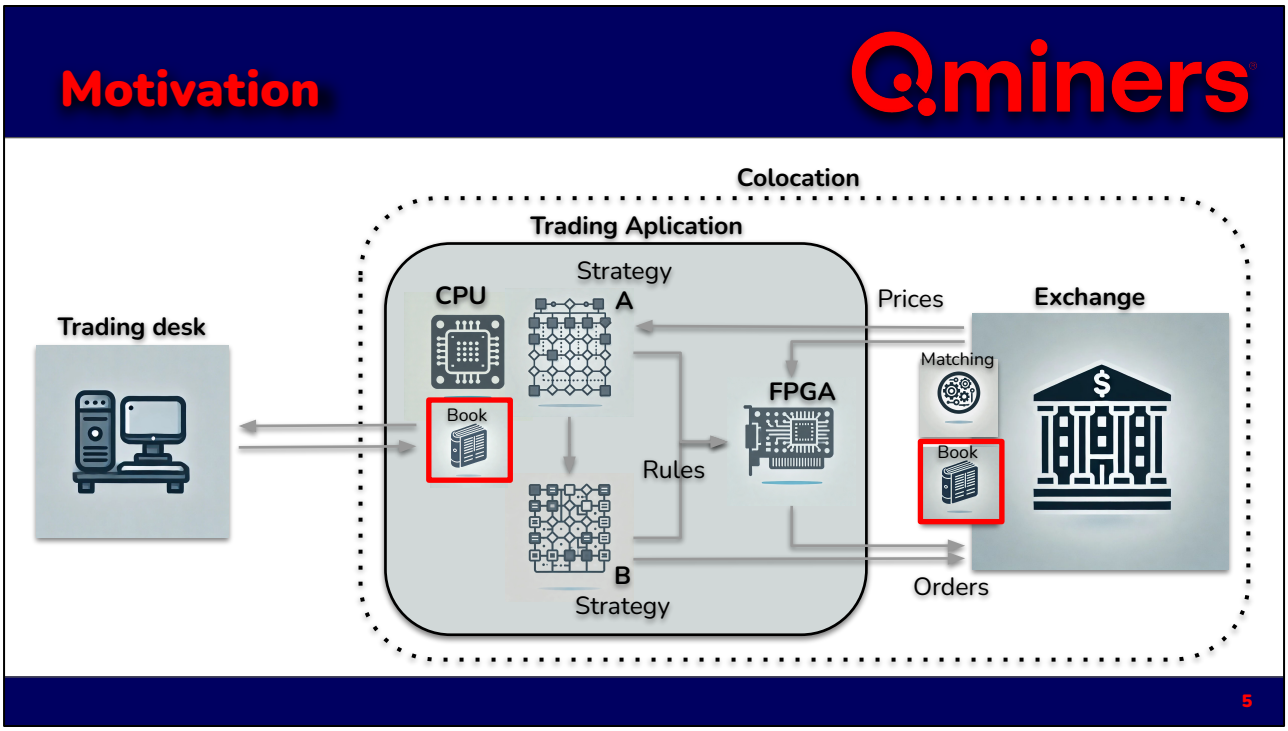
After that talk, I got some feedback from the audience saying I could’ve gone deeper - that they'd be interested in real production examples, not just isolated toy cases.
So this year, that’s exactly what I’m going to do.



This year, we're picking up where we left off - but with a slightly different approach.
Instead of covering a broad range of topics, I've chosen **one specific data structure**, and we'll try to tune it in every possible way - layout, cache locality, branching, SIMD… the full detail.
So rather than going wide, we're going deep - all the way down to the level of instructions and nanoseconds.

Now, just so no one gets scared off - let me add a quick disclaimer:
What I'll be showing today mostly comes from our C++ team, and even there, it's for very specific high-performance cases.
Most of our codebase is pretty normal C++ code (except for a certain level of hygiene - well designed memory management), but definitely not fine-tuned to this level of detail.
Also our Python team and analysts work on very different kinds of problems, at a very different level of abstraction.

But today I'll focus on the kind of development I know best - C++ and especially on the techniques I believe are broadly useful in any system where performance really matters.



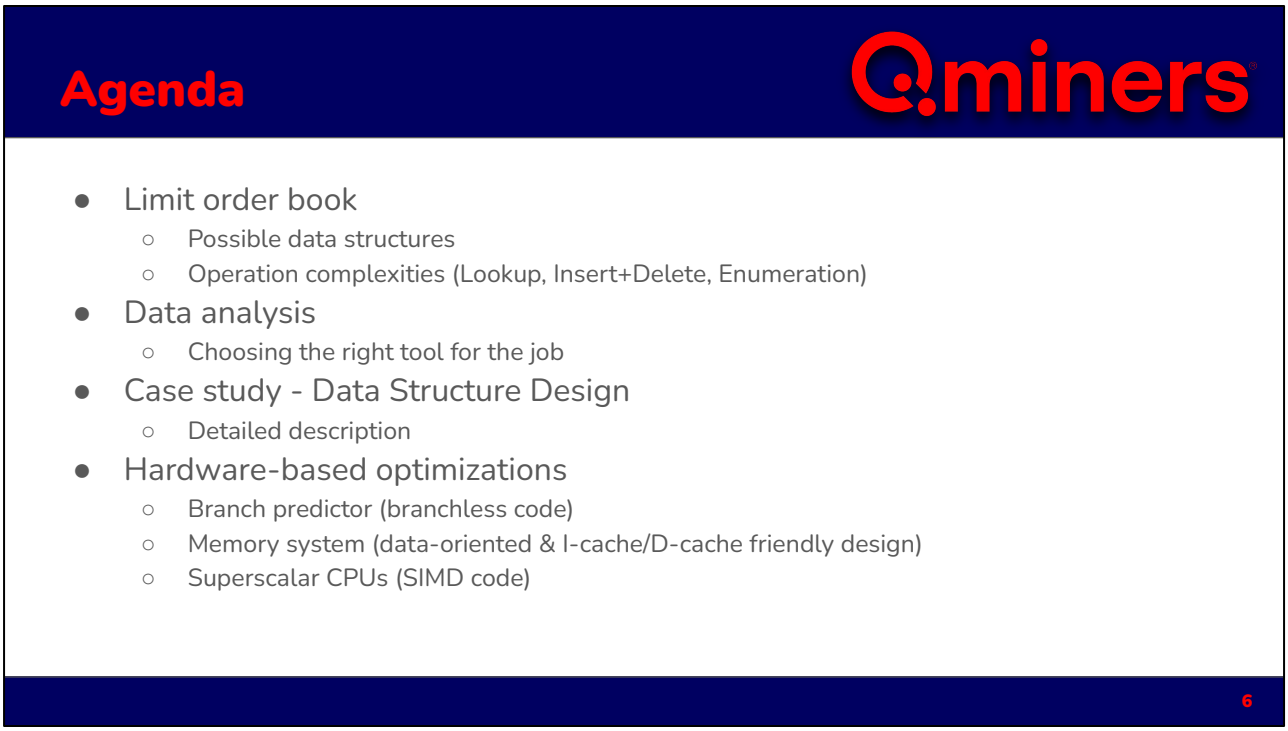
Let me start with a quick motivation - **why speed matters** in our field.

This diagram shows a simplified view of a **modern trading system**. On the right, we have the **exchange**, which you can think of as a central server that receives orders from different traders, maintains the **order book**, and uses a **matching engine** to pair up buy and sell orders. The results are then shared with all participants. On the left is the **trading application**, which acts like a **state machine**. It processes updates from the exchange and maintains its own internal view of the world - including its own copy of the order book.
Inside this app, one or more **strategies** are running, designed to react as fast as possible to market changes - by modifying, canceling, or submitting new orders.

Now, even if the trading app is colocated with the exchange - meaning it runs in the same datacenter - there's always a **delay**. Why? Because information is limited by the **speed of light** - roughly 1 nanosecond per 30 cm. That delay means your internal order book is **always slightly outdated**. The faster your software reacts, the more accurate your view of the market, and the more competitive your strategy. The diagram also shows that **some parts of the logic** can run outside the CPU - on **dedicated hardware** like **FPGAs**, which can react in **tens or hundreds of nanoseconds**, compared to several microseconds on CPU.

Finally, on the far left, there's the **workstation** used for supervision - required by regulator. From here, operators can tune strategy parameters or shut everything down in an emergency.
But in reality, the trading app is **fully autonomous**. If there's a bug, it can lose money **much faster than a human can react**.

So - the **order book** lives both inside the exchange and inside our trading application. It's the central **data structure** that represents the market state - and it's exactly what we're going to **analyze and optimize** in today's talk.

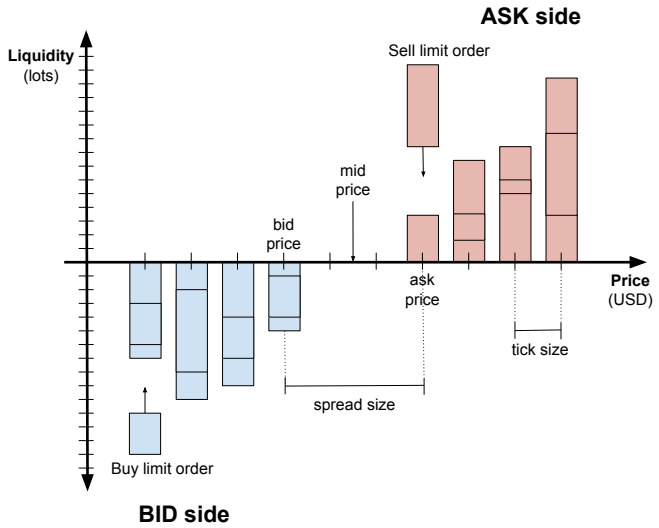


So let's take a look at today's agenda. We're going to use the **Limit Order Book** as our example.
But **my goal is broader:** I want to focus on designing data structures that are both performance-oriented and well-aligned with modern hardware.

1) We'll start with an **overview of** possible existing **data structures**.
2) Then we'll move into our **data analysis**. We'll start by looking at a real trading data and use it to build a **realistic benchmark**.
3) Then we'll dive into the **actual case study**, breaking down each design step for the **limit order book** and showing how we anchor it in reality.
4) Finally, we'll wrap up with **hardware-based optimizations**.
   We'll talk about how to make our code more **predictable** and **branchless**, lay out data to fit well into the **instruction and data caches**, and use the **prefetcher** and **SIMD instructions** effectively.

By the end of this presentation, you'll have a reusable playbook for designing complex data structures that can be applied **not just in finance, but in any performance-critical field**.

Limit Order Book



- Record of limit orders
  - Single security (contract)
- Limit orders
  - Price (discrete - tick size)
  - Quantity (discrete - lots)
- Two sides
  - Bid (buyers)
  - Ask (sellers)
- Prices sorted
  - Best price - highest bid
  - Best price - lowest ask

All right, let's start with the basics. What you see here is a **Limit order book**, which is essentially the **core data structure in electronic trading**. You can think of it as a live, **constantly updated list of buy and sell orders** for a single security - be it a **stock**, **futures contract,** or some other traded asset.

You can see that there are two sides:

- On the **left**, the **bid side** - buyers saying, *'I want to buy this much at this price.'*
- On the **right**, the **ask side** - sellers saying, *'I want to sell at this price.'*

Each bar here shows a **limit order** - an instruction to buy or sell **only** if a certain price is met.

- **Prices are discrete** - they move in steps called **ticks**, and the smallest price change is called the **tick size** - like one cent, or a fraction of a cent.
- **Orders are sorted** first by **price**, and then by **arrival time** at the same price level. That's called **price-time priority** - better price wins, and for ties, the first order in gets executed first.

Note that this is a view of just **a single contract** (e.g. a single futures expiration). In reality there are many of those. They can be kept separately, or all mixed in a single storage. That is an implementation/engineering choice.

Data Structure overview

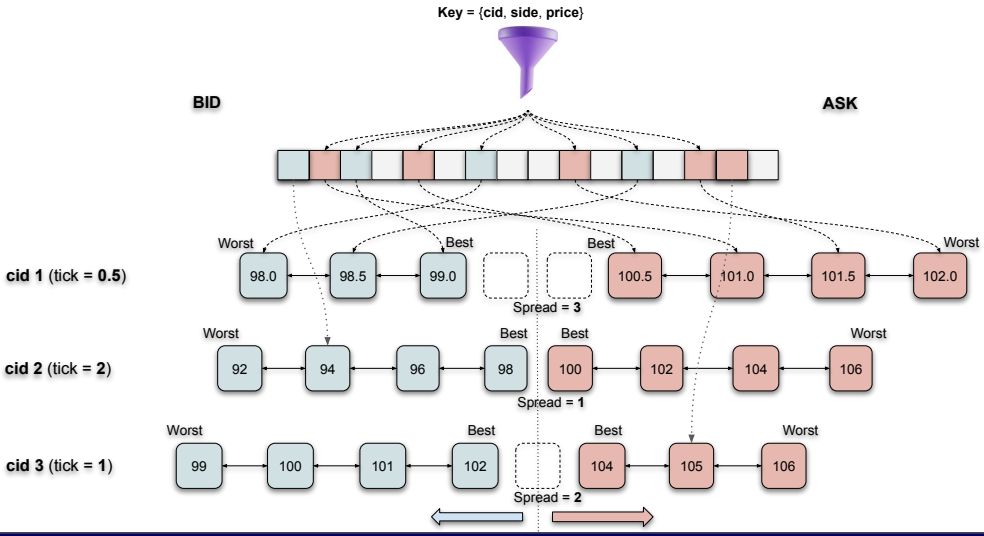
Qminers

Structure	Lookup	Ins/Delete	Notes
Hash + sorted list	O(1)	O(L)/O(1)	Fast lookup & delete, linear insert, node based content leads to pointer chasing during enumeration
RB/AVL/B+ trees	O(log N)	O(log N)	Balanced depth, deterministic lookup times, easy next/prev, cache unfriendly (except B trees)
Splay trees	O(log N)	O(log N)	Variable memory locality, Self-adapts to hotspots
Sorted vector	O(log N)	O(N)	Excellent locality, Fantastic for small-mid N
Buckets + Bitset	O(1)	O(1)	High mem if U large, excellent locality Top-tier latency if U reasonable

So on this slide, we're looking at a few common data structures you might consider for implementing an order book, each with its own pros and cons.

- If you go with a **hash map and a sorted list**, you'll get **fast lookups and deletes**, only insertions and enumeration can be a bit slower.
- Then you have **balanced trees** like Red-Black or AVL trees, which give you **reliable log-time performance**, but they're not always the best for cache usage.
- **Splay trees** may be a solid choice because of their self-adaptability, but may be too heavy for our purpose.
- **Sorted vectors** have a **great locality properties** and so are really friendly for the modern hardware, but have higher lookup and update asymptotic complexity.
- And if raw speed is your top priority and you're not worried about memory use, then using **Buckets with a Bitset** can give you really low latency.

Hash Lookup & Sorted Lists

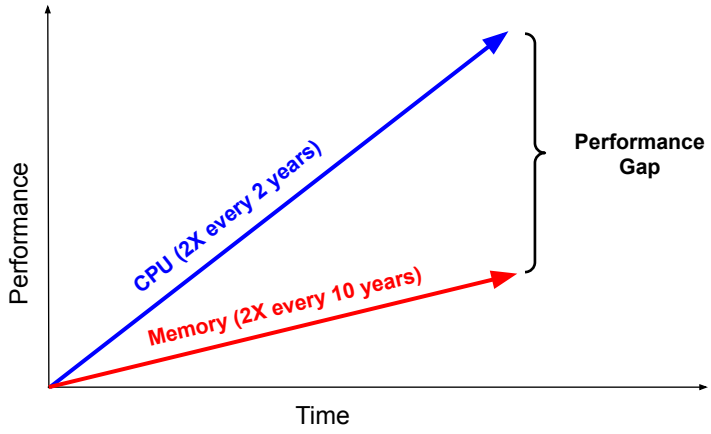


So let's take a closer look at the **hash lookup and sorted list combination**.
It's a nice hybrid approach that gives us really fast lookups using a **hash table**, and then we maintain order with a **sorted linked list**.
The hash table lets us jump directly to the right price level quickly, and from there, we can easily update or remove orders.
The trade-off is that when we're traversing the list, we might run into some **pointer chasing** because the nodes aren't stored contiguously in memory, which isn't the most cache-friendly choice. Also **insert must find next and previous nodes** to connect the new item to the linked list.
Still, it's a solid approach if you want that **balance of fast lookups and maintaining order**.

RAM is not flat

Qminers

- Cache hierarchy (3 layers)
  - **Level 1** ... 1 ns
  - **Level 2** ... 7 ns
  - **Level 3** ... 20 ns
  - **RAM** ... 100 ns
- CPU ideal throughput
  - **20 instr./ns**
    (1 Core * 4 IPC * 5GHz)
- L3 cache miss penalty
  - Up to **1:2000** !



10

To better understand why **linked list traversal is slow**, let's take a moment to look at **what memory latency really means in practice**.

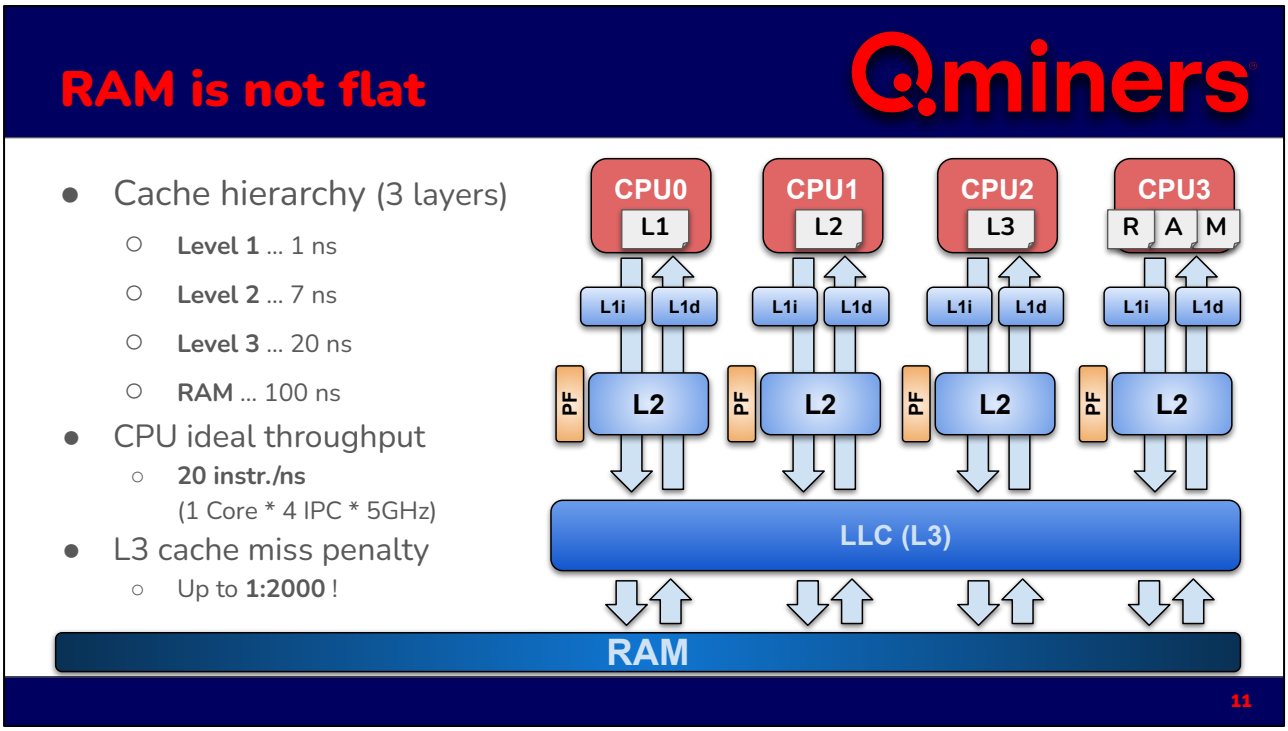
Yes - RAM stands for *Random Access Memory*, but in reality, memory isn't one big flat space. It's a **multi-level hierarchy**.

The key problem is that **processor speeds have increased much faster than memory speeds**. Over the years, CPUs have improved by about **60% per year**, while memory access speeds have improved by **less than 10%**.

This mismatch means that the processor often ends up **waiting for data**, which becomes a major bottleneck and limits overall system performance.

Modern systems try to **hide this latency** using multiple levels of cache. But every time you make a **random memory access**, like when chasing pointers in a linked list, you pay for it - because you may need to go through **several layers of cache** before you get your data.

And that's exactly what we want to avoid in performance-critical code.



We start at the top of the memory hierarchy with **L1 cache** - it's the fastest and sits closest to the CPU.
Then comes **L2**, **L3**, and finally **RAM**, which is about **100 times slower than L1**.

In the animation you'll see in a moment, each level has a different access latency - and if we want high performance, we need to take that into account.

This brings us to a key concept: the **principle of locality**.

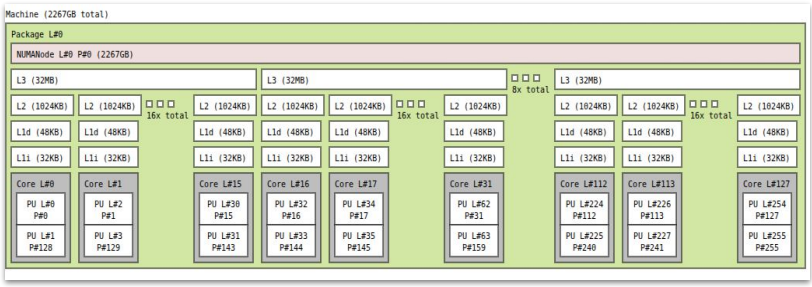
- **Temporal locality** means we often access the same value - or something recently used - again.
- **Spatial locality** means that if we access a certain memory address, we're likely to access nearby addresses soon after.

Every memory access, even when we read just a single byte, actually **loads an entire cache line** - typically **64 bytes** on most systems. And if your access pattern is **predictable** - say, linear or with a fixed stride - then the hardware can **guess where you're going next** and pre-load that data into cache ahead of time.
That logic is called the **prefetcher**.

And this is exactly why **data-oriented design** - where you shape your structures around how data moves through memory - can make your code run **dramatically faster**.

RAM is not flat

- Cache hierarchy (3 layers)
  - **Level 1** … 1 ns
  - **Level 2** … 7 ns
  - **Level 3** … 20 ns
  - **RAM** … 100 ns
- CPU ideal throughput
  - **20 instr./ns**
    (1 Core * 4 IPC * 5GHz)
- L3 cache miss penalty
  - Up to **1:2000** !



To make things more concrete - here's the machine we used for the benchmarks in this talk.

It's an **AMD EPYC 9745** with **128 cores** and 2TB of memory (no NUMA - all accessible from any of the cores). Each group of **16 cores shares a 32 MB L3 cache** - that's the last-level cache you see repeated in the diagram. Of course, every core also has its own **L1d cache** (around 48 KB) and **L2 cache** (about 1 MB).

The whole machine has around **2 TB of RAM**, and in our setup, it runs as a **single NUMA node** - which means we avoid cross-socket latency.
But still, L3 is **shared within core groups**, so if we're optimizing for latency, it makes sense to:

- **pin threads** to specific cores,
- and sometimes even **disable some cores**, to give the active ones **more L3 cache** to themselves.

That gives both the **prefetcher** and the **cache system** the best chance to work effectively.

And yes - this is the kind of hardware we run our code on every day (also all benchmarks presented here are measured on this machine).
So if this environment sounds interesting to you, feel free to come talk to us after the presentation - we're always happy to chat, or even schedule an interview if you're curious about joining Qminers.

Perf (cache-misses) - enumeration

Qminers

```
$ perf stat --control="fifo:/tmp/perfctl,/tmp/perfack" \
  -e "cycles:u,instructions:u,branches:u,branch-misses:u,cache-references:u,cache-misses:u" ./benchmark
Events disabled
Context: contracts=100, elements=21, lookups=1000000, enumerations=1000000, erases=125000, avgBest=3.75%, avgAny=47.63%
flat_book<int32_t, BigVector>:
  200*2*21 elements creation in 1 ms.
  200*2*1000000 best keys lookups in 1271 ms. (avg=0.79)
  200*2*1000000 any keys lookups in 1301 ms. (avg=10.00)
Events enabled
Events disabled
  200*2*1000000 levels enumerated in 4551 ms. (avg=210.00)


 Performance counter stats for './benchmark':
    16,805,640,277      cycles:u                                                (83.34%)
    52,211,180,857      instructions:u                    #    3.11  insn per cycle              (83.33%)
     9,402,154,488      branches:u                                              (83.33%)
         1,025,319      branch-misses:u                   #    0.01% of all branches             (83.33%)
    10,608,808,425      cache-references:u                                      (83.34%)
             8,106      cache-misses:u                    #    0.00% of all cache refs           (83.33%)

      10.145595968 seconds time elapsed
```


Let’s zoom in on just one operation: **enumerating the book from best to worst**. Using the linux perf measurement tool (using control pipes showed in a bonus slide below), I bracket only the enumeration loop, so the counters you see are for that section alone - no init, no teardown.

First, the `flat_book` version. Enumeration is a **straight, contiguous walk** over two arrays (keys then values).
That’s exactly what caches and the hardware **prefetcher** like.
The result: **almost no cache misses** and very high **IPC** - the CPU streams through the data.

Perf (cache-misses) - enumeration

Qminers

```
$ perf stat --control="fifo:/tmp/perfctl,/tmp/perfack" \
  -e "cycles:u,instructions:u,branches:u,branch-misses:u,cache-references:u,cache-misses:u" ./benchmark
Events disabled
Context: contracts=100, elements=21, lookups=1000000, enumerations=1000000, erases=125000, avgBest=3.75%, avgAny=47.63%
hash_book<int32_t, BigVector>:
  200*2*21 elements creation in 1 ms.
  200*2*1000000 best keys lookups in 1763 ms. (avg=0.79)
  200*2*1000000 any keys lookups in 1607 ms. (avg=10.00)
Events enabled
Events disabled
  200*2*1000000 levels enumerated in 8629 ms. (avg=210.00)
LevelBook load factor: 0.6835381235251038

 Performance counter stats for './benchmark':
    31,902,390,545      cycles:u                                                (83.33%)
    36,611,551,617      instructions:u                    #    1.15  insn per cycle              (83.33%)
     9,002,411,279      branches:u                                              (83.34%)
         1,261,783      branch-misses:u                   #    0.01% of all branches             (83.34%)
    18,800,169,498      cache-references:u                                      (83.34%)
        56,569,894      cache-misses:u                    #    0.30% of all cache refs           (83.33%)

      15.022031156 seconds time elapsed
```


Now compare that to the **hash map**. A hash table is great for **random O(1) lookups**, but it's **unordered**.
To enumerate in price order you end up following **linked lists** and jumping around memory.
The prefetcher can't predict those addresses, so you see a **lot of data-cache misses** and a much lower IPC.
Wall-clock time goes up too.
Note that even though the number of instructions went significantly down (from **52B** down to **36B**), it takes significantly more cycles as the number of instructions per cycle drops from **~3** to **~1**.

I call this a **horizontal traversal** example - we're walking along the **price axis**.
It shows how layout decides whether the CPU streams or stumbles from miss to miss.
On the next slide we'll look at the **vertical traversal** problem - misses caused by **hash collisions** inside the table itself.

Perf (cache-misses) - enumeration

```
$ perf stat --control="fifo:/tmp/perfctl,/tmp/perfack"
  -e "cycles:u,instructions:u,branches:u,branch-misses
Events disabled
Context: contracts=100, elements=21, lookups=1000000,
hash_book<int32_t, BigVector>:
  200*2*21 elements creation in 1 ms.
  200*2*1000000 best keys lookups in 1763 ms. (avg=0.7
  200*2*1000000 any keys lookups in 1607 ms. (avg=10.0
Events enabled
Events disabled
  200*2*1000000 levels enumerated in 8629 ms. (avg=210
LevelBook load factor: 0.6835381235251038

 Performance counter stats for './benchmark':
    31,902,390,545      cycles:u
    36,611,551,617      instructions:u
     9,002,411,279      branches:u
         1,261,783      branch-misses:u
    18,800,169,498      cache-references:u
        56,569,894      cache-misses:u

      15.022031156 seconds time elapsed
```

```
        template <typename Level>
        inline const Level* LevelTableLinkedListImpl<Level>::getTopLevel(bool isSell) const
        {
        return m_level[isSell];
  0.24    mov       (%rcx),%rax
        for (bool isAsk : {false, true}) {
        auto level = book.get_best(cid, isAsk);
        for (; level; level = level_next(level))
  1.20    test      %rax,%rax
  0.08  ↓ je        7e
          data16    cs nopw 0x0(%rax,%rax,1)
  0.02    xchg      %ax,%ax
        avg += level_val(level);
  0.05  70:  vaddsd     0x30(%rax),%xmm0,%xmm0
        return m_next;
 44.44    mov       0x8(%rax),%rax
        for (; level; level = level_next(level))
 11.98    test      %rax,%rax
  5.94  ↑ jne       70
  7e:   mov         0x8(%rcx),%rax
  0.10    test      %rax,%rax
  0.01  ↓ je        9e
          nop
        avg += level_val(level);
  0.01  90:  vaddsd     0x30(%rax),%xmm0,%xmm0
 29.74    mov       0x8(%rax),%rax
        for (; level; level = level_next(level))
  1.47    test      %rax,%rax
  4.47  ↑ jne       90
        for (int cid : ctx.contracts) {
  9e:   add         $0x4,%rdx
  0.10    cmp       %rdx,%rsi
        ↑ jne       50
        for (size_t e = 0; e < ctx.enumerations; ++e) {
  a7:   add         $0x1,%r8
  0.01    cmp       %r10,%r8
        ↑ jne       3e
  b0:   vmovsd      %xmm0,0x8(%rsp)
        }
```


Here is a detailed visualization of hotspots in **perf report**.
You can see concrete **instructions responsible for most cache misses**.
And it is clear that they are the ones dereferencing the **m_next** pointer inside the linked list.

Hash Collisions

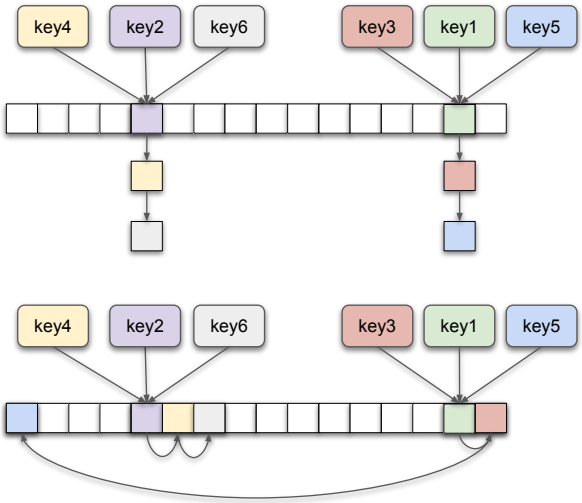
Qminers

Close addressing (separate chaining)

- `std::unordered_map` (stable allocators)
- Cache unfriendly → slow
- Easy to build price level chains

Open Addressing (linear probing)

- `boost::unordered_flat_map` `(unstable allocators)`
- Cache friendly → much faster
- Hard to build price level chains



Now, earlier we mentioned that **pointer chasing** becomes a performance issue not just in price level traversal, but also in the **hash table itself** - especially when you hit **collisions**, meaning two or more keys land in the same bucket.

Traditionally, the standard approach was **closed addressing**, or **separate chaining** - like in `std::unordered_map`. Colliding entries are stored in a linked list, or chain, attached to each bucket. That's simple, and it gives us **stable memory addresses**, which is great for linking entries together - for example, to build **price level lists**.

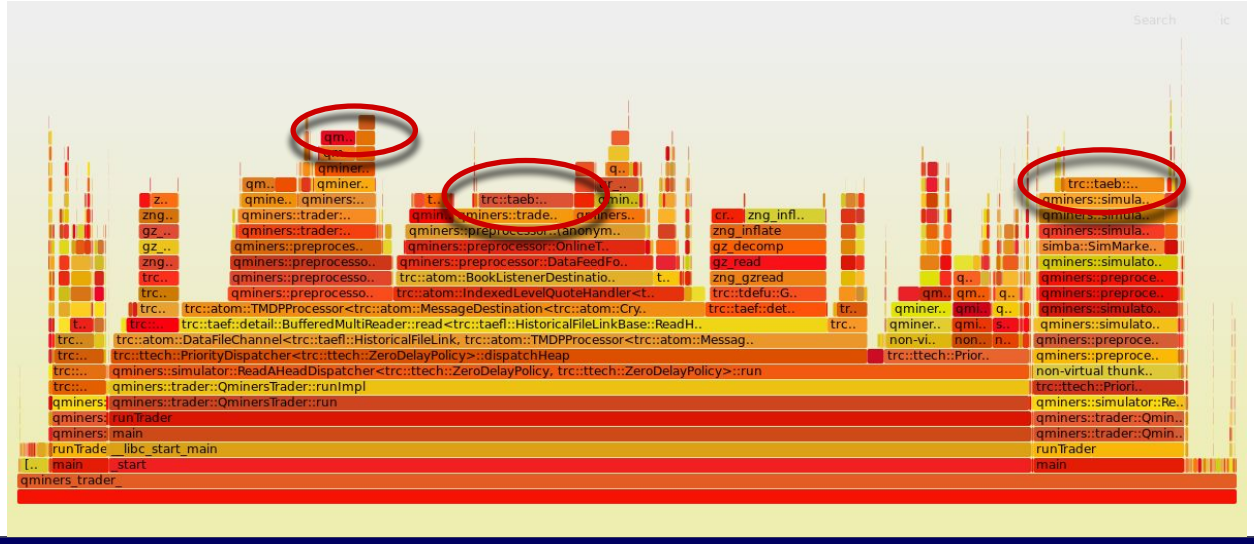
But it's **cache unfriendly** - because every lookup or traversal jumps through pointers, leading to slower performance on modern CPUs.

More modern hash tables - like `boost::unordered_flat_map` - use **open addressing**, typically with **linear probing**. Here, colliding entries are just stored in the next available slot in the array itself. That's much more **cache friendly**, and in practice, much faster.

However, the downside is: the entries move around in memory - so we lose that stability. That makes it **hard to form things like best-to-worst price level chains**, because you can't just maintain pointers between price levels anymore.

So in a way, we're forced back into that old-world model of **node-based containers**, which isn't ideal for high-performance trading systems.

Linux perf (FlameGraph visualization)



Here's a real-world example where all the things we've been talking about - data structures, memory locality, and cache behavior - came together... but unfortunately, **not in a good way**.

We were using **hash tables with collision chains**, as I showed earlier - with a preconfigured bucket array size.
Then we onboarded a **new market** with **very fine price granularity** and **high volatility**, and we made a mistake: We **didn't adjust the hash table size** accordingly.

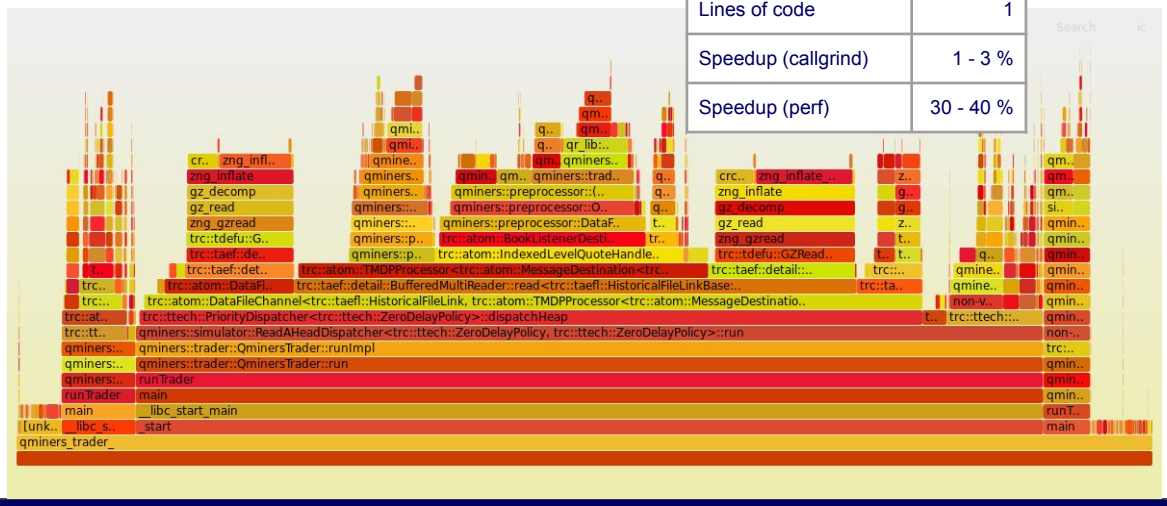
The result? Collision chains became long, and our **lookups degraded from O(1) to something closer to linear scans** through linked lists.
Combined with modern hardware - where **random memory jumps** mean **cache misses** and **branch mispredictions** - the performance dropped **dramatically**.

On this slide, you can see a **flame graph** - the red areas show exactly where the hotspots were. And yes, they're all related to **hash table lookups** in our order book logic.

Linux perf (FlameGraph visualization)

Qminers

Lines of code	1
Speedup (callgrind)	1 - 3 %
Speedup (perf)	30 - 40 %



How did we fix it? Simple: we **resized the hash table** to reduce the load factor - and that solved it. On this slide, the new flame graph confirms the improvement. And in the bonus slides, I've included detailed `perf` measurements showing how performance **gradually degrades as load factor increases**.

What's interesting is that our **Callgrind profiling** didn't catch it. It showed only a small increase in instruction count - just a few percent. But once we started measuring **actual runtime** and **clock cycles** using `perf`, it was immediately clear: **30-40% slowdown** in key parts of the code.

Access Pattern Analysis

- Focus on **real market data**, not toy workloads
- Log every book access over a full day
- Look at the hottest books and most liquid contracts (representative load)
- Turn findings into a **synthetic benchmark** that **mirrors reality**
- **Optimize for the observed pattern**
- Measure wall-clock **time** and **counters**

So far, we've looked at different implementation options, and we've examined in more detail where a typical approach starts to fall short.

But before we even begin choosing a new data structure, we first need to understand **how the order book is actually used** during real-world trading.

So I did the most logical thing - I **logged every access to the order book over the course of a full trading day**.

We'll analyze this real data and use it to build a **synthetic benchmark** that closely mirrors actual market behavior.

That way, we can test different implementations under **realistic conditions**.

Because the goal isn't to optimize for some theoretical scenario - it's to optimize for **what actually happens on the market**.

Analysis - overview

Qminers

Log stats (single product, single day, log size ... 20GB, 316M messages, 5K msg/s average, 200K msg/s peek):

```
$ wc ~/tmp/flat-book.log
   316659738  2216618137 20794534012 ~/tmp/flat-book.log
```

Message preview (**time,** instance **pointer**, **method name**, **cid**, **side**, **price**, resulting **position**, map **size**):

```
$ grep flat_book ~/tmp/flat-book.log | head -n5000000 | tail -n5
 23:20:12.586800 0x16456da0 flat_book::get_best(0, false, nan) ... pos:0, size:0
 23:20:12.586800 0x16456da0 flat_book::get_best(0, true, nan) ... pos:0, size:0
 23:20:12.586800 0x16456da0 flat_book::get_best(1, false, nan) ... pos:0, size:0
 23:20:12.586800 0x16456da0 flat_book::get_best(1, true, nan) ... pos:0, size:0
 23:20:12.586800 0x16456da0 flat_book::get_best(2, false, nan) ... pos:0, size:0
```

Split by instance (dominant one has 230M accesses):

```
$ grep "flat_book" ~/tmp/flat-book.log | cut -d" " -f2 | sort | uniq -c | sort -nr
 232475369 0x267fdc88
  41143323 0x1b70eca0
  24476560 0x19e6b160
  10130623 0x16456da0
   2940421 0x16e0e108
   2886269 0x163d3fc8
   2607158 0x2692c538
```


So I have instrumented the system and **logged every book access** for a sample day on one product: about **20 GB** of logs and **316 M** events.
Each record captures the **book instance**, **method**, **contract**, **side**, **price**, and the **resulting level index** and **table size** - number of **price levels**.

There are several book instances; we'll focus on the **hottest one** with **~230 M** accesses.
From this we will hopefully see real **access patterns** - best level jumps, neighbor walks, random lookups - which will guide the structure we choose.

Analysis - called methods

Qminers

Split dominant instance by method (*`get_best()`* gets called 150M times!):

```
$ grep ^0x267fdc88 ~/tmp/flat-book.log | cut -d":" -f5- | cut -d"(" -f1 | sort | uniq -c | sort -nr
154294356 get_best
 54179875 get
 21753740 next
  2186814 get_or_create
    30375 get_or_create<insert>
    29975 erase
      234 get_or_create<alloc>
```

Note that **other instances** may have different histogram:

```
$ grep ^0x19e6b160 ~/tmp/flat-book.log | cut -d":" -f5- | cut -d"(" -f1 | sort | uniq -c | sort -nr
24336970 get
  104946 get_or_create
   17322 erase
   17238 get_or_create<insert>
      84 get_or_create<alloc>
```


Here's what we actually call most.

On the hottest book instance, `get_best` **dominates** - about **150M calls** - then `get` (price lookup) and `next` (a horizontal traverse).
Inserts/erases are **two orders of magnitude lower**.

So the workload is **read-heavy**, with a huge bias to **best-of-book** and short **neighbor walks**.

But it's not uniform: another instance flips the order - there, `get` wins.
Takeaway: our structure must make **best-of-book O(1)** and neighbor traversal **sequential/cache-friendly**, while still giving fast **price→level** lookups.
That's exactly why the **flat, contiguous layout** could fit this pattern

Analysis - most liquid contract

Most liquid contract:

```
$ head ~/tmp/flat-book-74.log
0x267fdc88 flat_book::get_or_create<alloc>(74, false, -10) ... pos:0, size:1
0x267fdc88 flat_book::get_best(74, false, -10) ... pos:0, size:1
0x267fdc88 flat_book_level::next(74, false, -10) ... pos:1, size:1
0x267fdc88 flat_book::get_best(74, false, -10) ... pos:0, size:1
0x267fdc88 flat_book_level::next(74, false, -10) ... pos:1, size:1
0x267fdc88 flat_book::get_best(74, false, -10) ... pos:0, size:1
0x267fdc88 flat_book_level::next(74, false, -10) ... pos:1, size:1
0x267fdc88 flat_book::get(74, false, -10) ... pos:0, size:1
0x267fdc88 flat_book::get(74, false, -10) ... pos:0, size:1
0x267fdc88 flat_book::get(74, false, -10) ... pos:0, size:1
```

Method call histogram (`get()` called **1.2M** times):

```
$ cut -d":" -f3- ~/tmp/flat-book-74.log | cut -d"(" -f1 | sort | uniq -c | sort -nr
1246418 get
 905319 get_best
  54632 get_or_create
   5970 next
    208 get_or_create<insert>
    194 erase
      6 get_or_create<alloc>
```


Note that **there are many contracts** in the log.
Now we split the data by contract and focus only to the most liquid one.

For the most liquid contract (**cid = 74**) - `get` is a dominant method - called **1.2M times** (followed by `get_best` with **900K calls**).

Analysis - lookup pos & sizes

Qminers

Histogram of lookup positions and map sizes for most liquid contract:
(very small map and most lookups are at best positions)

Lookup position and table size histograms:

```
$ grep "::get(" ~/tmp/flat-book-74.log | cut -d" " -f6 | sort | uniq -c | sort -nr
 920516 pos:0,
 168932 pos:1,
  66116 pos:2,
  20891 pos:3,
  17726 pos:4,
   8107 pos:5,
   5165 pos:6,
   3945 pos:7,
   3670 pos:14,
   3568 pos:8,
   3516 pos:12,
   3504 pos:15,
   3458 pos:13,
   3427 pos:11,
   3420 pos:9,
   3375 pos:10,
   2711 pos:17,
   1209 pos:16,
   1151 pos:19,
   1148 pos:20,
    863 pos:18,
```

```
$ grep "::get(" ~/tmp/flat-book-74.log | cut -d" " -f7 | sort | uniq -c | sort -nr
 312377 size:18
 307253 size:20
 290918 size:17
 165739 size:19
 148873 size:21
   7135 size:15
   5782 size:16
   5121 size:14
   1153 size:13
   1000 size:12
    304 size:7
    255 size:2
    147 size:1
    138 size:3
     66 size:4
     42 size:6
     34 size:11
     30 size:5
     20 size:8
     17 size:10
     14 size:9
```

95% of lookups are at first three levels:

```
$ for I in 0 1 2 3 4 5; do short=$(grep "::get(" ~/tmp/flat-book-74.log |
cut -d" " -f6- | grep -c "pos:[0-$I]"); long=$(grep "::get("
~/tmp/flat-book-74.log | cut -d" " -f6- | grep -vc "pos:[0-$I]");
result=$(echo "scale=2; 100*$short / ($short + $long)" | bc); echo "$I
... $result %"; done
0 ... 73.85 %
1 ... 89.56 %
2 ... 94.95 %
3 ... 96.63 %
4 ... 98.05 %
5 ... 98.70 %
```


Let's zoom in on `get(price)` - one of our hottest paths.

1) The **lookup position** distribution is far from uniform. Most lookups are at first three levels. That fits intuition - most action is at the best.
2) The **book size** is tiny - typically **~18–21** levels. That's great news: inserts and shifts are cheap at this scale.
3) The accumulated histogram is far highly skewed: **~74%** hit the best level, **90%** within top 2, **95%** within top 3.

Takeaway: We can beat a hash table by using a **flat, contiguous layout** with **direct index** and a **fast path** for the top 1-3 levels.

Benchmark - lookup keys

Qminers

Key generator

```
std::random_device rd;
std::mt19937 gen(rd());
```

Random keys

```
std::vector<price_t> result(size);
std::uniform_real_distribution<> dis(0.0, elements);
for (size_t i = 0; i < size; ++i)
  result[i] = static_cast<price_t>(dis(gen));
```

Hot keys

```
const std::vector<double> weights{ 920516, 168932, 66116, 20891, 17726, 8107, 5165, 3945, 3568, 3420, 3375,
                                   3427,   3516,  3458,  3670,  3504, 1209, 2711, 1151, 1148,  863};
std::discrete_distribution<> dis(weights.begin(), weights.end());
for (size_t i = 0; i < size; ++i)
  result[i] = static_cast<price_t>(dis(gen));
```


To keep the lookup test fair, I generate the **keys once** with a single RNG and reuse the **same sequence** for every container.

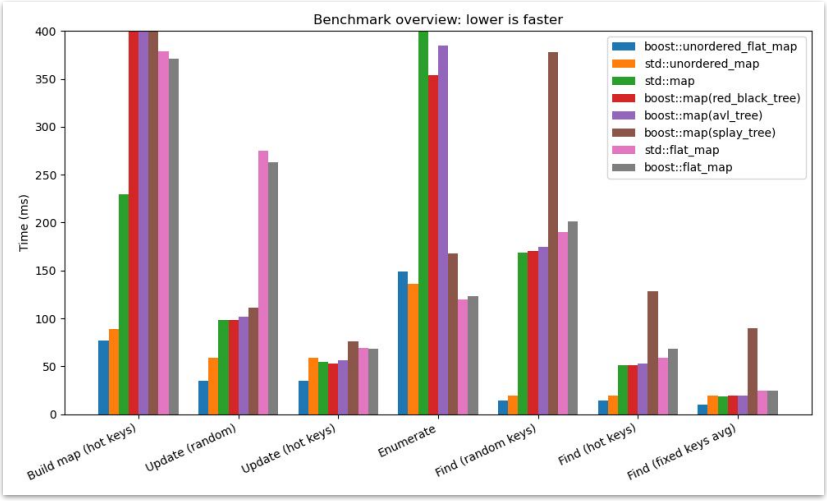
Two modes:

1) First, **random levels** using a uniform distribution over N prices - the worst case, no locality.
2) Second, **hot levels** using a discrete distribution **seeded from our real histogram** - retrieved directly from the log file.
   That way we compare each structure both in a **uniform** world and in the **skewed, real** world.

The runs are reproducible with a **fixed seed**, and the key generation happens **outside the timed region**.

Benchmark - baseline (3rd party maps)

- 3rd party maps
  - type ... `array<byte, 1024>`
  - size = 21
- 10M operations
  - Create
  - Insert/Delete
    - Random keys
    - Hot keys
  - Enumerate
  - Lookup
    - Random keys
    - Hot keys
    - Fixed keys



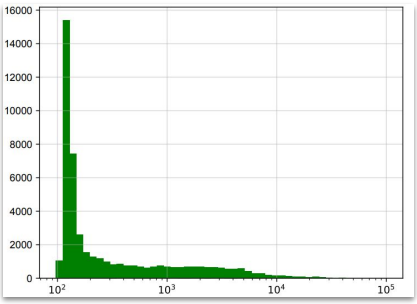
In order to establish a baseline I have tested different maps - things like `boost` and `std unordered_map`, `flat_map`, and `std::map` - to see which one handles a small order book best.

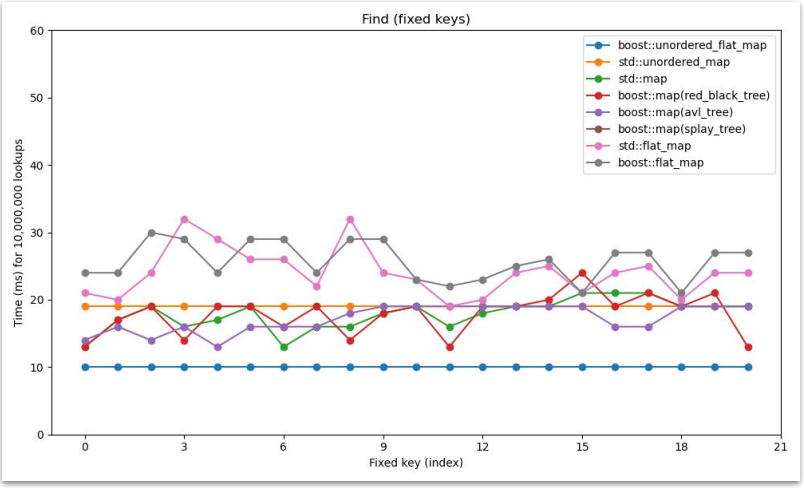
1) For both random and hot key **lookups**, `unordered_flat_map` **was fastest** because it has **O(1)** hash lookup an keeps data contiguous which **avoids pointer chasing**.
   Tree-based `std::map` and flat maps just **aren't as fast due to** more complicated **binary search** (`splay_tree` lookups are really slow).
2) For **enumeration** - which means traversing through the list in order, `flat_map` was the best because it uses memory efficiently.
   Node-based maps are just **not as fast**.

The unordered maps look fine for enumeration, but note that the **enumeration results here are skewed**.
In a real order book, we need to walk the book from best to worst price, but these maps enumerate in **their own internal order** (that is why they are called "**unordered**").
So while this shows raw speed, in reality, we'd have to add extra logic to get the order we need (e.g. use the `unordered_node_map` and **form linked lists in the expected order**).

Benchmark - baseline (3rd party maps)

- 3rd party maps
  - type … `array<byte, 1024>`
  - size = 21
- 10M lookups
  - Fixed keys (0 … 20)





Alright, here's the a **special case of this benchmark**.

We're **hitting the each individual index millions of times**, everything's in the cache, and we're just measuring the **raw instruction speed**.

1) **Open-addressing maps** like `unordered_flat_map` **come out on top** (and its performance is really stable).
2) **Chaining** `unordered_map` **is significantly slower** (but provides nice stable performance) , and both **tree-based maps** and `flat_map` **do more comparisons** during the binary lookup.

And also keep in mind, these are best-case numbers with everything hot in the cache and perfectly predicted branches, so it is just a **measurement of a pure instruction throughput**. In real life, we also have to consider what happens **when data isn't perfectly cached** and how well are the branches predicted.
So then you measure a histogram, so single data structure visualization would lion something like this … *[see the green histogram]*.

If there is enough time, I have plenty of additional benchmarks at bonus slides showing effects of **cache-misses** and **branch-misses**.

Data Oriented Book Design

Takes modern HW into account

- **D-cache** friendly ("flat" storage)
  - Temporal locality
  - Spatial locality (**Cache line** and **Prefetcher** aware code)
- **I-cache** friendly (simple logic)
  - Indexing
  - Linear search
- **Branch predictor** friendly
  - Mostly branchless code
  - If branches, then predictable
- **Superscalar CPU** friendly
  - Integral keys
  - SIMD instructions

Aspire for best of both worlds (fast *lookup* and *enumeration*)

So let's talk about how we **design our structure** to really **fit modern hardware**.

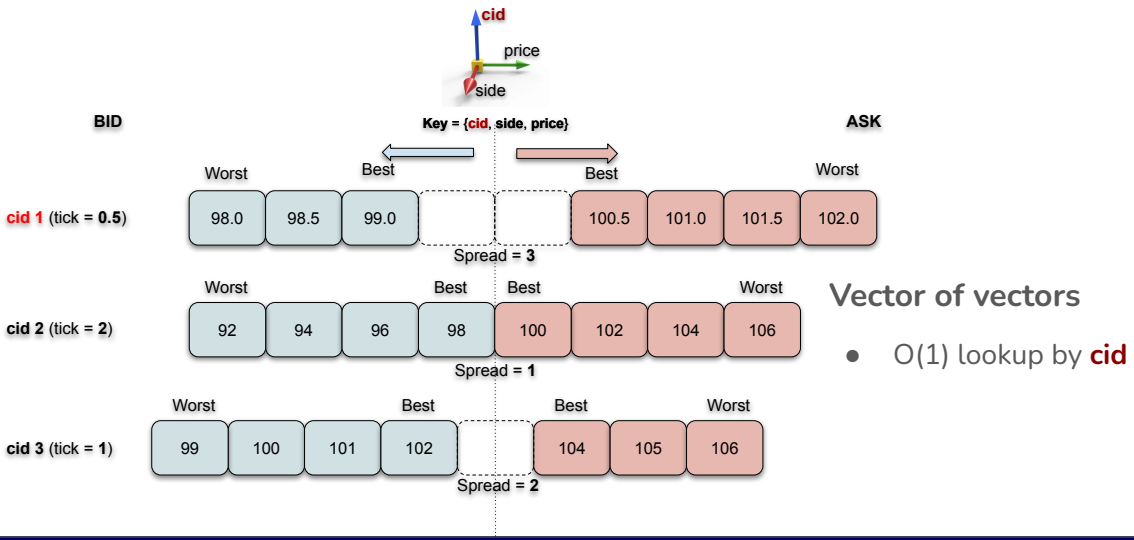
The idea is to get the best of both worlds:

1) On one hand, an `unordered_flat_map` **gives us those super fast O(1) lookups**, but it's unordered and the entries aren't stable.
   If we need a best-to-worst price list, we end up switching to something like an `unordered_node_map` and **start linking nodes together**.
   That means **pointer chasing** and cache misses.
2) On the other hand, a `flat_map` **keeps everything nicely sorted by price**, so going through it is super fast.
   But lookups become a **branchy binary search** and a bit **slower**.

Our situation is special: we only have about **20 levels** and most of the action is at the **top 1 to 3 levels**.
So instead of using a general-purpose map, we design for the hardware.
We keep everything flat and **contiguous**, **sort by price**, and calculate the index with **simple arithmetic**.
That way it's **D-cache friendly**, easy for the **prefetcher**, and we could get enumeration as fast as a `flat_map` and lookups as fast as a **hash map**.

So that sounds like a plan.

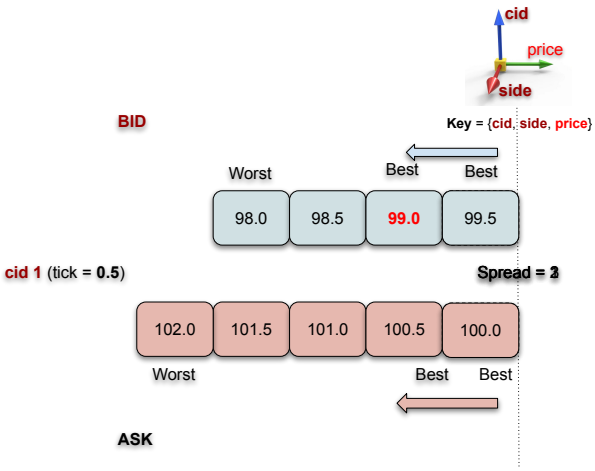
Book Design - Contract selection



So instead of using a hash table with collisions, we keep price levels in nice, **flat arrays**.
For each **contract**, the keys are just a **sorted array of prices**.
We can use the contract ID as a direct index - no hashing needed - so it's **O(1)** access.

The big win is **locality**.
Updates hit the same small area of memory, so we **stay in L1 or L2 cache**, and the prefetcher works great.
It also makes things cleaner: settings are per contract, bookkeeping is simpler, and we can even **shard by contract for concurrency** if we want.
In short, we go from a big global structure to a per-contract view, which matches how the data arrives and how the CPU likes it.

Book Design - Level insert/delete



Avoid shifting the vector

- Inserts/Deletes are expensive
- Push/Pop back is cheap
- Move best levels towards the end

29

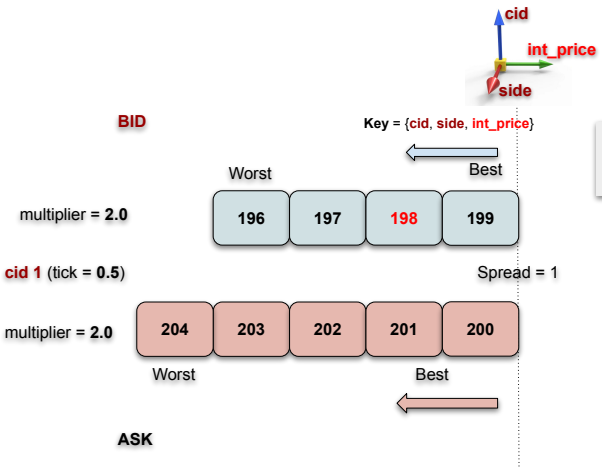
The next step is to **split the price array to two halves**, so that each contract just has a **BID array** and an **ASK array**, all neatly laid out.

And here's another neat trick with vectors:
Instead of dealing with expensive inserts at the front, we arrange the vector so that the **best price level is always at the back**.

For **bids, prices go up from left to right**, so the highest bid - the best price - is at the end.
For **asks, prices go down from left to right**, so the lowest ask - the best price - is at the end.
That means when the best price changes, we just push or pop from the back, which is super fast and cache-friendly.

If we occasionally have to insert somewhere in the middle, that's fine.
The book is small - around 20 levels - and any changes **deeper in the book aren't so latency-sensitive anyway**.
Most of the speed game happens at the top levels, so that's where we focus our optimizations.

Book Design - Integral Prices



Utilize discrete prices

- Divide by tick size (precalculate inverse)

```
int32_t price2key(const auto& contract, double px) const {
    return fast_round(px * contract.invtick);
}
```

- `int32_t` keys (data cache friendly)
- `int16_t` keys can be considered
  - 32 keys in cache line
  - 64K levels might not be enough for volatile markets
- Dense books
  - we can directly use it for **O(1)** indexing (`idx = (price - base) / tick`)
- Sparse books - sorted keys search
  - Binary ... **O(log N)**
  - Sequential ... **O(N)**

30

So here's another idea: since prices move in **discrete ticks**, we don't need **floating-point numbers**.
Instead, we **convert prices into integers** by multiplying by the inverse of the **tick size** for that contract.
That way, we avoid any floating-point weirdness, and our comparisons are fast and exact.

We typically use a **32-bit integer** for the price index, which is big enough and still fits nicely in the cache (16 keys in a 64 byte cache line).
Using integers makes everything predictable and cache-friendly.

1) Now, if the market's price range is tight, and **the book is dense** - we can just use **direct indexing**.
   That means we calculate the index from the integer price and **use it as an array offset** - super fast and efficient **O(1)**.
2) If the price range is wider, and **the book more sparse** - we use a sorted flat vector and do a quick **linear scan O(N)** if it's really small - like 20 levels - or a **binary search O(logN)** for deeper books.

Overall the price lookup sounds like a more general approach than direct indexing, so we are going to analyze that in following slides. In short, using integers for prices keeps everything exact, predictable, and really **friendly to the CPU and cache**.

Book Design - Side selection

Bid vs. Ask price ordering

- Different types
- Template bloat
- Ternary operator → Unpredictable branch

```
reverse_flat_map<int32_t, T, std::less> bid;
reverse_flat_map<int32_t, T, std::greater> ask;

const T* get(key_type key) const {
  const auto& c = contracts[key.cid];
  const int px = price2key(c, key.price);
  return key.isAsk ? c.ask.get(px) : c.bid.get(px);
}
```


So here's another nice trick that simplifies our design and keeps things efficient.

We want the **best level at the back** for both bids and asks.
That means we need opposite sort orders: **bids sorted one way** and **asks the other**.

The straightforward way is to use two different comparators - **`std::less` for bids** and **`std::greater` for asks** - but that leads to two different map types, more template code, and a branch that's **hard to predict** because **bid/ask traffic is effectively random**.

Book Design - Side to Sign

Bid vs. Ask price ordering

- Same types
- Less template bloat (I-cache friendly)
- Branchless code (Predictor friendly)

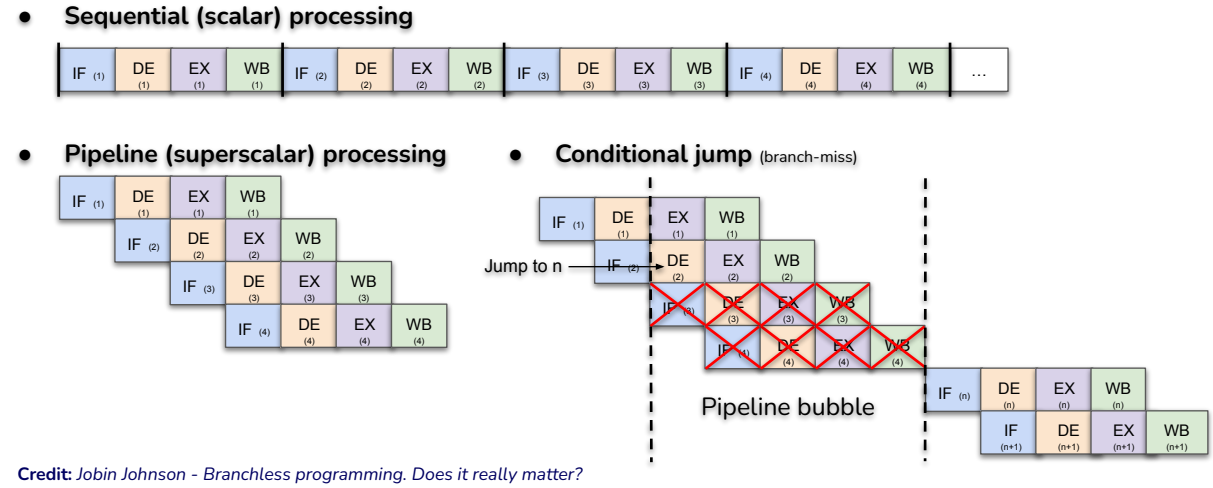
```
std::array<reverse_flat_map<int32_t, T>, 2> sides;

const T* get(key_type key) const {
  const auto& c = contracts[key.cid];
  const int px = price2key(c, key.price);
  const int sign = 2 * key.isAsk - 1; // +1 ask, -1 bid
  return c.sides[key.isAsk].get(sign * px);
}
```


Instead, we **encode the side into the key itself**.
We turn the price into an integer tick index and then multiply by a sign: +1 for asks and -1 for bids.
That way, both sides end up in the same map type, sorted in the same direction, with the best price at the back.

The result?
We have **no unpredictable branches**, just one map type for both sides, a smaller binary, and a single direction for the **prefetcher** to follow
(less code is **better for instruction cache**).

CPU Pipeline - branch miss



Credit: *Jobin Johnson - Branchless programming. Does it really matter?*

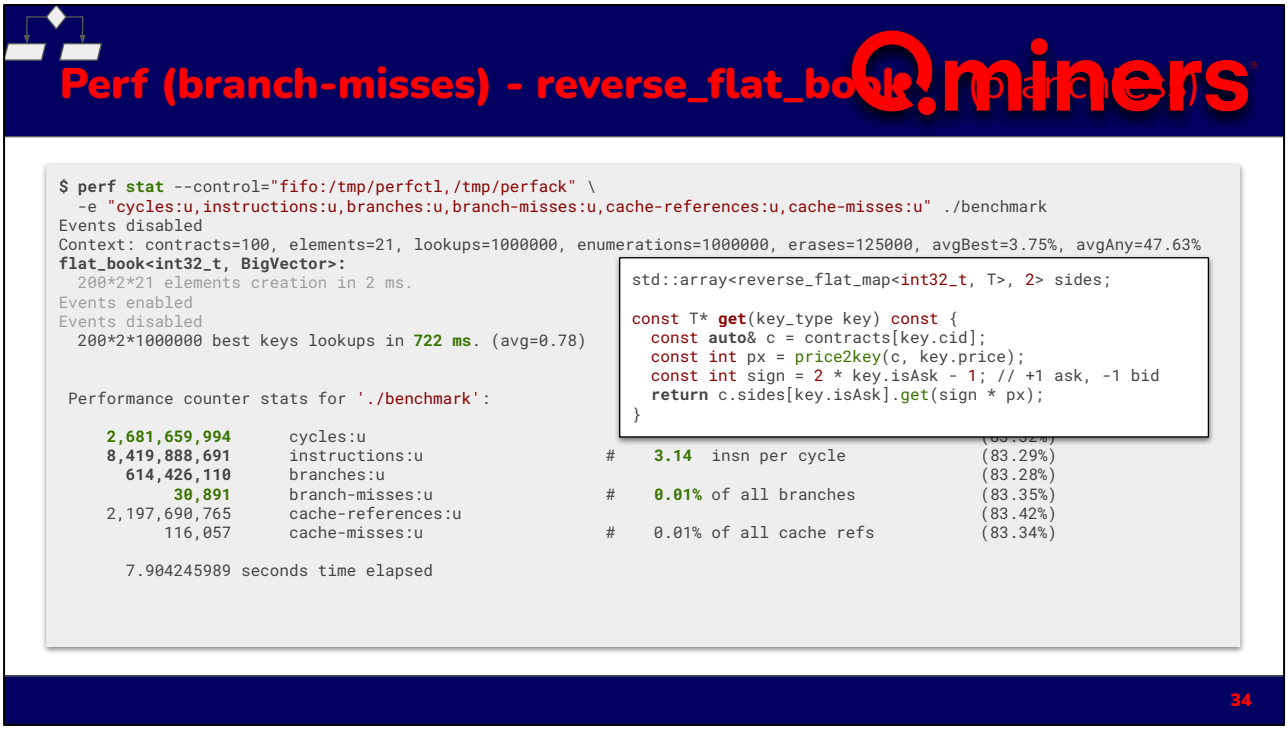
33

Now let's take a look at **why it's so important to avoid branches** - especially the kind that are **hard to predict**, like the one we saw earlier. This brings us to a really interesting topic: **how modern processors execute instructions**. An instruction isn't executed as one atomic step. Internally, it's broken down into **micro-operations**, and it goes through several phases: **Fetch, Decode, Execute, Write Back**.

What you see here is a **naive model**, where all micro-operations are processed **sequentially** - one after the other. That's how it worked, for example, in the **80286 processor from 1982**. The first simple three-stage **pipeline** came in the **80386** in 1985. But modern CPUs are **superscalar**. That means the pipeline is **wide**, and the processor can execute **multiple instructions in parallel**, even within **a single thread**.

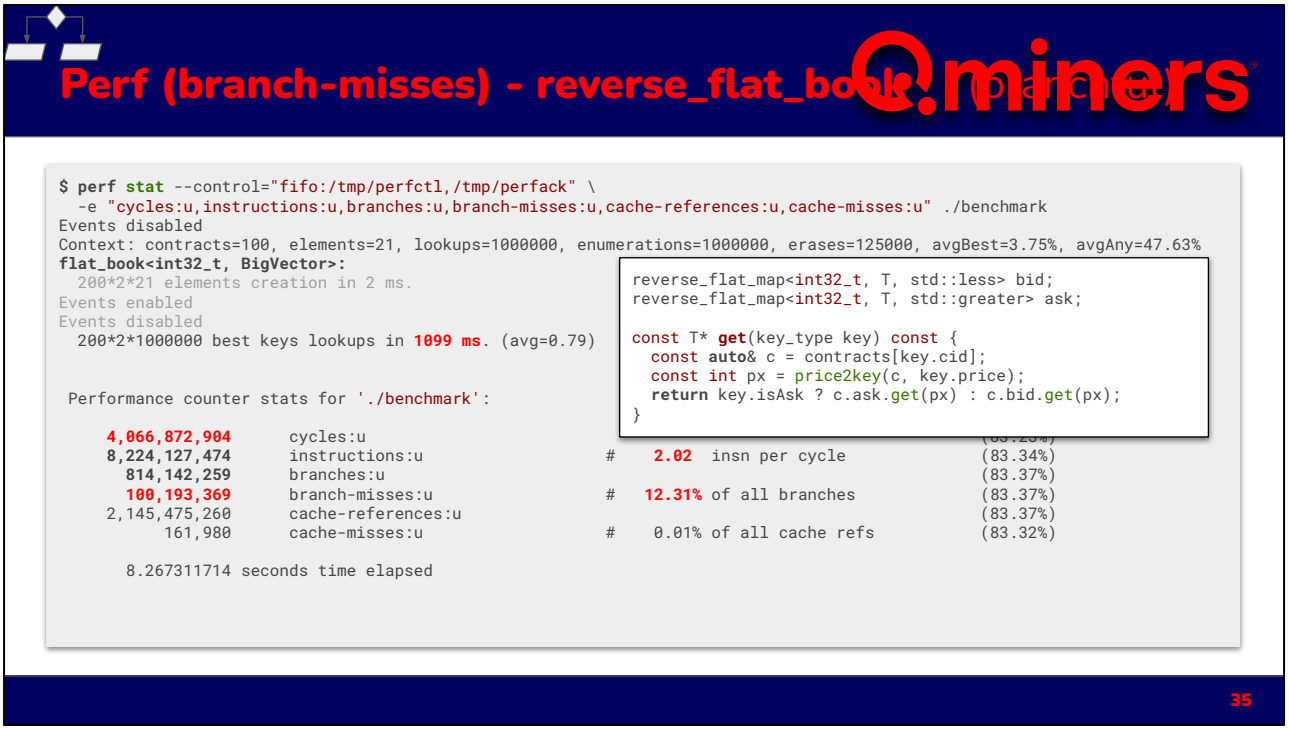
The problem comes with **conditional jumps**. After a jump, the CPU doesn't know which instruction comes next until the condition is evaluated. Waiting for that would stall the pipeline - so instead, CPUs use **speculative execution**. They **guess** the outcome of the branch and start executing the instructions **ahead of time**.

If the guess is **correct**, great - everything continues smoothly. But if the **branch predictor fails**, the pipeline has to be **flushed** - all speculative work is thrown away, and the CPU starts over. This creates what's called a **pipeline bubble**, and it can easily cost **10 or more CPU cycles**.
The component in the CPU that handles this guessing is, unsurprisingly, called the **Branch Predictor**.
And this is exactly why unpredictable branches can have a **massive impact** on performance - they break the flow of modern superscalar execution.



Now let's look at this particular source of slowdown - **branches**, not caches.

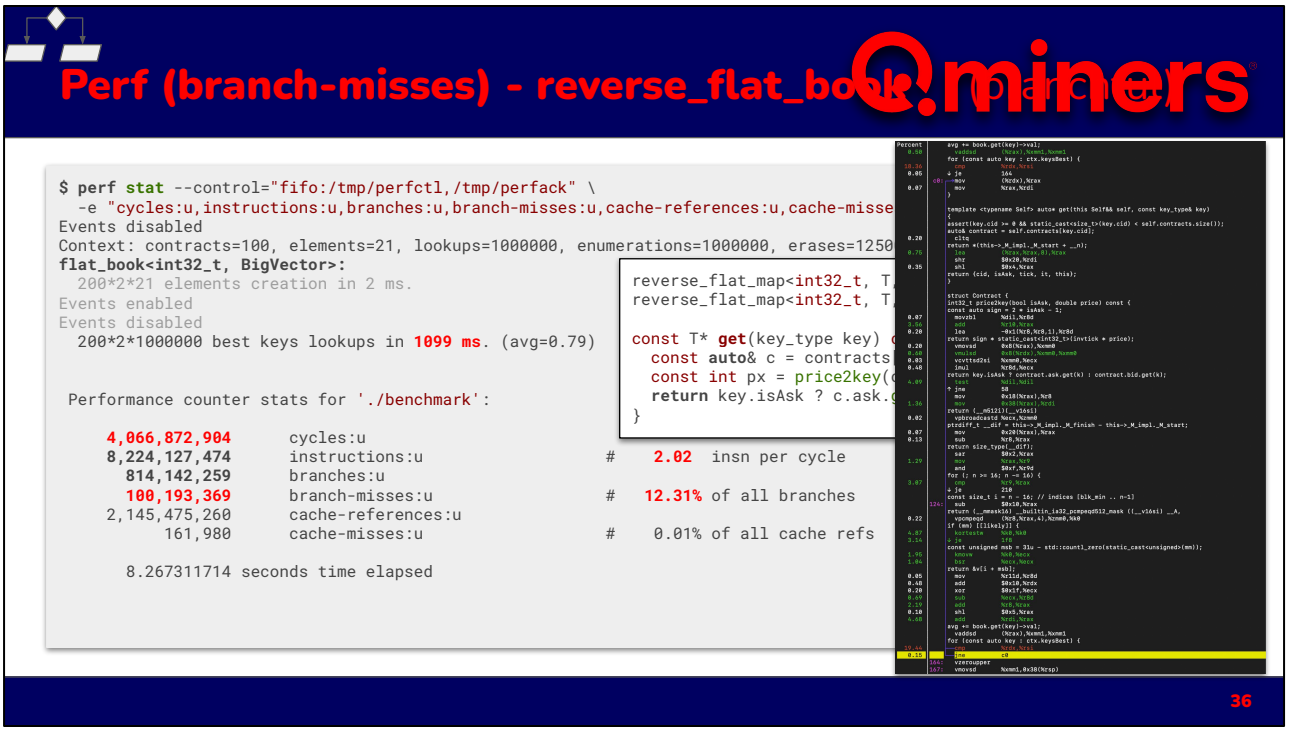
I'm comparing two versions of the lookup. In the **branchless** version we keep a single `reverse_flat_map` for both sides.
We convert price to an integer tick and multiply by **±1** depending on side, so both BID and ASK are stored in the same order.
Then we index an `std::array` with `isAsk`.
There's **no** `if`, it's compact, and the instruction cache is happy.



In the **branchy** version we have two different map types - `std::less` for bids and `std::greater` for asks - and a **ternary** to choose between them.
In the benchmark the side is random, so the **branch predictor guesses wrong a lot**. Every miss means the CPU has to roll back speculation and refill the pipeline.

You can see the effect: the **instructions per cycle** drop from about **3.1 to 2.0**, the **cycle count** rises from roughly **2.7B to 4.0B**, and **branch-misses** shoot up by orders of magnitude.

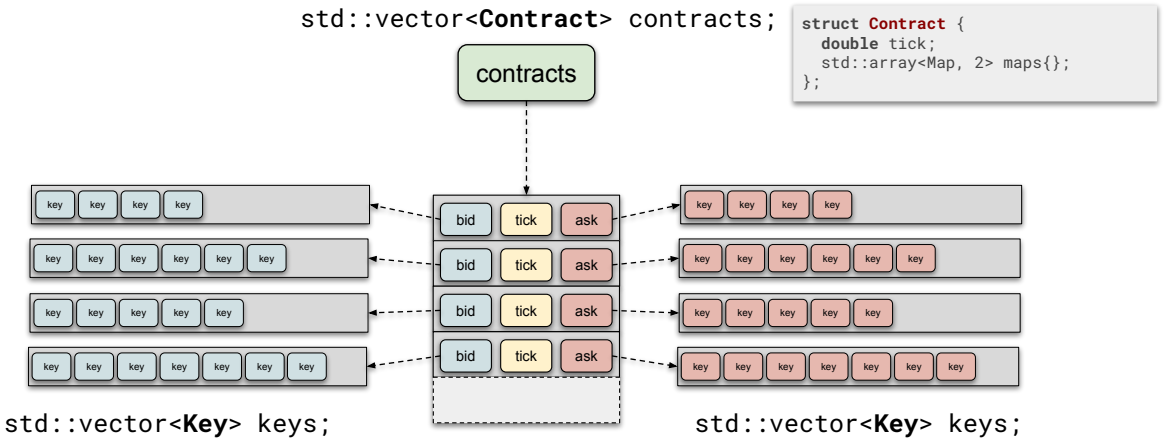
You can see that the branchful version has 200M more branches out of which 100M is missed - so that is exactly as expected (if the branch source is basically random, branch predictor fails to predict the correct branch is 50% of cases, just like when tossing a coin).



The **perf report** points right at the hotspot - the conditional jump on `isAsk` - and you can also see the **lookup code duplicated**, which adds pressure on the instruction cache.

The lesson is simple: if a branch is **unpredictable at runtime**, try to remove it - encode it into data or use arithmetic/array indexing.
Predictable branches are usually cheap on modern CPUs; **unpredictable** ones are not.

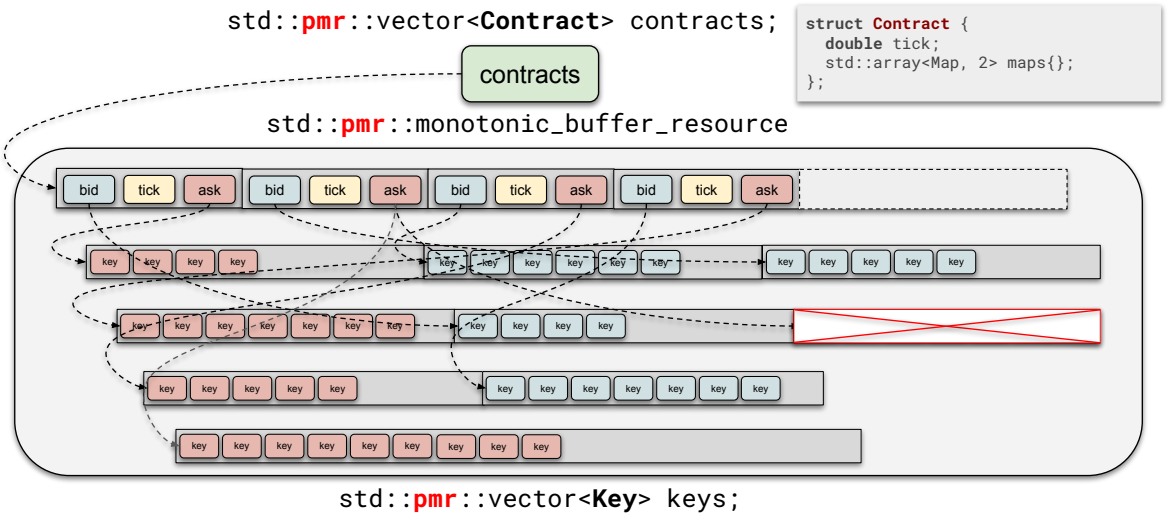
Book Design - Default Allocator



Alright, let's talk about how we handle memory.

In the **default world**, every `std::vector` uses the **global heap**.
That means each vector grows wherever it can find space, and we end up with memory scattered all over.
Every time a vector needs to grow, it has to go back to the global allocator, deal with thread safety, and potentially move things around.
It works, but it's not very cache-friendly and it's kind of noisy.

Book Design - Custom Allocator (PMR)



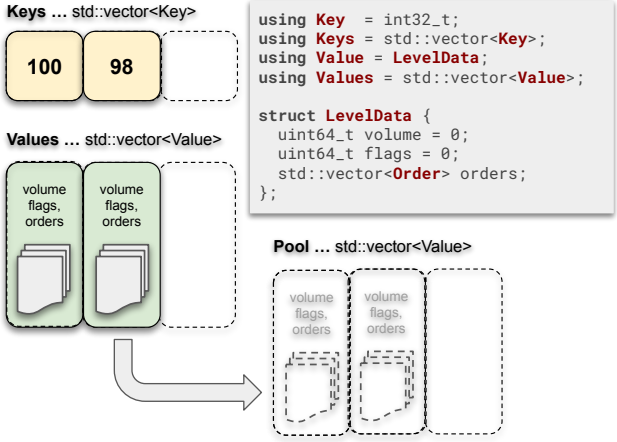

```
struct Contract {
  double tick;
  std::array<Map, 2> maps{};
};
```


Now, on the **right side**, we switch to **PMR** (polymorphic memory resources).
Think of it as giving the book keys their own little **private memory arena**.
We use a `std::pmr::vector` and a **monotonic buffer**, so all allocations come from one contiguous chunk of memory.
That means we get much better locality, fewer trips to the global allocator, and the prefetcher can do its job.

We do trade off a little flexibility because this memory doesn't get freed until we reset the arena, but the gains in predictability and cache performance are worth it.

Book Design - Preserve Values' capacity

`reverse_flat_map<Key, Value>`




```
using Key   = int32_t;
using Keys  = std::vector<Key>;
using Value = LevelData;
using Values = std::vector<Value>;

struct LevelData {
  uint64_t volume = 0;
  uint64_t flags = 0;
  std::vector<Order> orders;
};
```

```
bool erase(const Key& key, Pool* pool) {
  const auto pos = reverse_flat_map::find_pos(keys_, key);
  if (reverse_flat_map::found(keys_, key, pos)) {
    const auto vit = std::begin(values_) + pos;
    if (pool && pool->size() < pool->capacity()) {
      if constexpr (requires { vit->clear(); }) {
        vit->clear();
      } else {
        // TODO: consider using a static empty_ here
        *vit = {};
      }
      pool->push_back(std::move(*vit));
    }
    keys_.erase(std::begin(keys_) + pos);
    values_.erase(vit);
    return true;
  }
  return false;
}
```

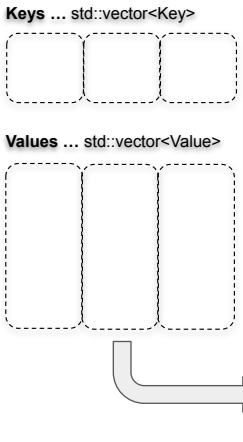

This is a small but meaningful optimization focused on **preserving memory capacity** - not just in the `keys_` and `values_` vectors, but also **inside each `Value` object** containing possibly nested containers.

Let's start with `erase()`:
When we remove a price level, we don't destroy the value. Instead, we **move it into a shared pool** - a simple `std::vector<Value>`. This pool can be reused across multiple books, for example between the **bid and ask** sides of contracts. But there's a subtle design question here: should we **reset the value** before putting it into the pool?

- If you're shaving nanoseconds in a closed system, maybe not.
- But if you want the map implementation to be **generic and reusable**, you don't want to risk leaking data - like old order IDs or flags - between reused values.

Book Design - Preserve Values' capacity

`reverse_flat_map<Key, Value>`




```cpp
using Key   = int32_t;
using Keys  = std::vector<Key>;
using Value = LevelData;
using Values = std::vector<Value>;

struct LevelData {
  uint64_t volume = 0;
  uint64_t flags = 0;
  std::vector<Order> orders;
};
```

```cpp
bool erase(
  const aut
  if (rever
    const a
    if (pool
      if constexpr
        vit->clear
      } else {
        // TODO: consider using a static empty_ here
        *vit = {};
      }
      pool->push_back(std::move(*vit));
    }
    keys_.erase(std::begin(keys_) + pos);
    values_.erase(vit);
    return true;
  }
  return false;
}
```

```cpp
vector& operator=(vector&& rhs) {
  clear();
  if (!rhs.empty() || capacity() < rhs.capacity())
    swap(rhs);
  return *this;
}
```

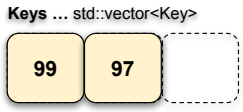

The obvious way to clear a value would be to assign an empty one: `*vit = {}`. But that's dangerous here. Why?
Because containers like `std::vector` (inside `Value`) in C++ **give up their capacity** when assigned via **r-value** - meaning we'd lose the very memory we were trying to preserve. So instead, we look for better options:

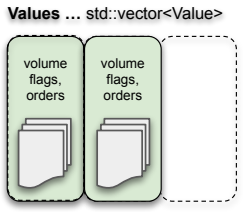
- If the type has a `clear()` method, we call that - it preserves capacity.
- Otherwise, we could assign a static empty instance using **copy assignment**, which **does** preserve capacity if implemented properly.

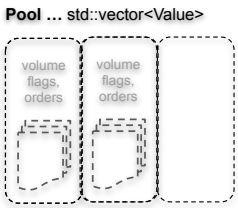
In the end, you can even detect this via `requires` and write a reset logic that fits each `Value` type. That gives us **safe reuse** without sacrificing performance or memory reuse.

Book Design - Preserve Values' capacity

`reverse_flat_map<Key, Value>`








```cpp
using Key   = int32_t;
using Keys  = std::vector<Key>;
using Value = LevelData;
using Values = std::vector<Value>;

struct LevelData {
  uint64_t volume = 0;
  uint64_t flags = 0;
  std::vector<Order> orders;
};
```

```cpp
template <typename Func> requires std::invocable<Func, Key, Value&>
Value& get_or_create(const Key& key, Func&& create_func, Pool* pool) {
  const auto pos = reverse_flat_map::find_pos(keys_, key);
  auto vit = std::begin(values_) + pos;
  if (!reverse_flat_map::found(keys_, key, pos)) {
    keys_.insert(std::begin(keys_) + pos, key);
    if (pool && !pool->empty()) {
      vit = values_.insert(vit, std::move(pool->back()));
      pool->pop_back();
    } else {
      vit = values_.emplace(vit);
    }
    create_func(key, *vit);
  }
  return *vit;
}
```


Now for the flip side - when we need to create a new price level.

The function `get_or_create()` checks if the level already exists; if not, we insert a new key and **reuse a `Value` from the pool** if available.
The nice thing is: this works with any `Value` type. The reused instance is **moved** back into the map's `values_` vector, and we immediately invoke a `create_func` **functor**, which initializes the object to its correct logical state.

This design gives us:

- No fresh allocation unless the pool is empty,
- Memory **locality preserved** (reused blocks tend to stay hot in cache),
- And safe, explicit initialization using a clean, generic interface.

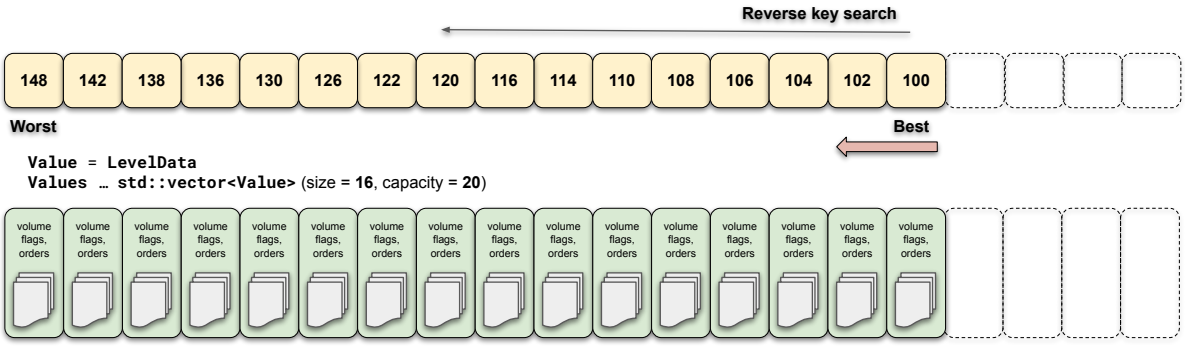
It's a small bit of code, but it shows the kind of **engineering detail** that pays off in high-performance systems - and still stays flexible enough for general use.

Book Design - Price Key Lookup

Qminers

`reverse_flat_map<Key, Value>`

`Key` … **`int32_t`**
`Keys` … **`std::vector<Key>`** (size = **16**, capacity = **20**)



So here's how we handle lookups in a really efficient way.

Instead of doing a **binary search**, we scan from the back of the book toward the worse prices.
Because most lookups hit the top levels, this **sequential scan** is **simpler**, **predictor friendly** and **usually faster** than binary search.

We also **separate keys from values** into two different vectors.
That way we can scan through just the keys at full speed and only load the values once we find what we need.
This keeps everything **cache-friendly** (16 keys per cache line) and very efficient.

Key Lookup variants

https://godbolt.org/z/E3avdf6qq

```
template <typename Key> size_t lower_bound_desc_bin(std::span<const Key> keys, Key key) noexcept {
  const auto it = std::lower_bound(keys.cbegin(), keys.cend(), key, std::greater<Key>{});
  return std::distance(keys.cbegin(), it);
}
```

https://godbolt.org/z/dd8c9x1nn

```
template <class It, class T, class Cmp> It branchless_lower_bound(It first, It last, const T& value, Cmp cmp) noexcept {
  for (auto length = last - first;  length > 0; ) {
    const auto half = length >> 1;
    first += cmp(first[half], value) * (length - half);
    length = half;
  }
  return first;
}
```

https://godbolt.org/z/195hWvch3

```
template <typename Key> size_t lower_bound_desc_rseq(std::span<const Key> keys, Key key) noexcept {
  const auto rit = std::find_if(keys.crbegin(), keys.crend(), [&](auto x){ return x > key; });
  return std::distance(keys.cbegin(), rit.base());
}
```

https://godbolt.org/z/P1E66T8Gx

```
template <typename Key> size_t lower_bound_desc_raw(std::span<const Key> keys, Key key) noexcept {
  const auto* k = keys.data();
  for (auto n = keys.size(); n > 0; --n) if (k[n-1] > key) return n;
  return 0;
}
```


Let's quickly compare three ways to find a key in our sorted array.

- First, we have **binary search** using `std::lower_bound`. It's **O(log N) on paper**, but for our small and top-heavy arrays, it's not always the fastest.
- Second, there is a simple `branchless_lower_bound` implementation for reference
- Fourth, we do a **reverse sequential scan** based on `std::find_if` starting from the best price. This matches our real-world usage and is often faster because it's more predictable even though it is **O(N) on paper**.
- Third, we have a simple **hand-written loop**. It does the same reverse scan, produces very nice and simple assembly, but it's not really faster than the standard algorithm.

In short, a **simple reverse scan** often **beats binary search** for our small dataset. And if you want to see the generated assembly, I've included **Godbolt links**.

Key Lookup variant (vectorized)

Qminers

https://godbolt.org/z/z4YxqsYdE

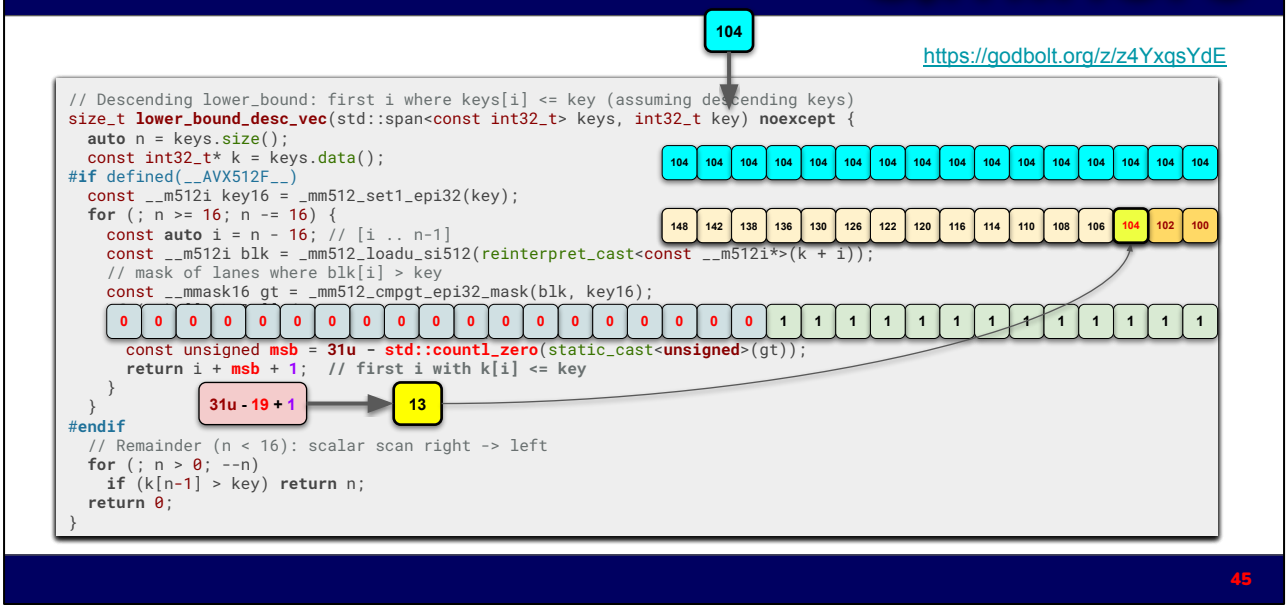
```
// Descending lower_bound: first i where keys[i] <= key (assuming descending keys)
size_t lower_bound_desc_vec(std::span<const int32_t> keys, int32_t key) noexcept {
  auto n = keys.size();
  const int32_t* k = keys.data();
#if defined(__AVX512F__)
  const __m512i key16 = _mm512_set1_epi32(key);
  for (; n >= 16; n -= 16) {
    const auto i = n - 16; // [i .. n-1]
    const __m512i blk = _mm512_loadu_si512(reinterpret_cast<const __m512i*>(k + i));
    // mask of lanes where blk[i] > key
    const __mmask16 gt = _mm512_cmpgt_epi32_mask(blk, key16);
    if (gt) [[likely]] {
      // pick the lane nearest to the tail (highest set bit)
      const unsigned msb = 31u - std::countl_zero(static_cast<unsigned>(gt));
      return i + msb + 1;  // first i with k[i] <= key
    }
  }
#endif
  // Remainder (n < 16): scalar scan right -> left
  for (; n > 0; --n)
    if (k[n-1] > key) return n;
  return 0;
}
```


Here's a fast path, which is my favourite right now. It's the same **reverse sequential** idea, but **vectorized**.

- With **AVX-512**, we load **16 int32 keys at once** from the tail (**`_mm512_loadu_si512`**), compare them to the **broadcasted key**, and get a **bitmask** of 'greater than'.
- Then we find the first position where the sequence **drops less than or equal to the key** by counting leading zeros in that mask.
- Because most lookups are at the top, the loop almost never needs a second iteration; it returns after the **first 16-key probe**.
- Any remainder (<16) falls back to a tiny scalar tail.

Loads are **unaligned** (we scan from the end), which is fine on modern CPUs; but note that we typically **touch two cache lines**.
We also compile a scalar fallback when AVX-512 isn't available (AVX2/SSE or plain loop).
I have created the AVX2 alternative as well (see the **Godbolt link**).

Key Lookup variant (vectorized)



Here's the fast path, which is my favourite right now. It's the same **reverse sequential** idea, but **vectorized**.

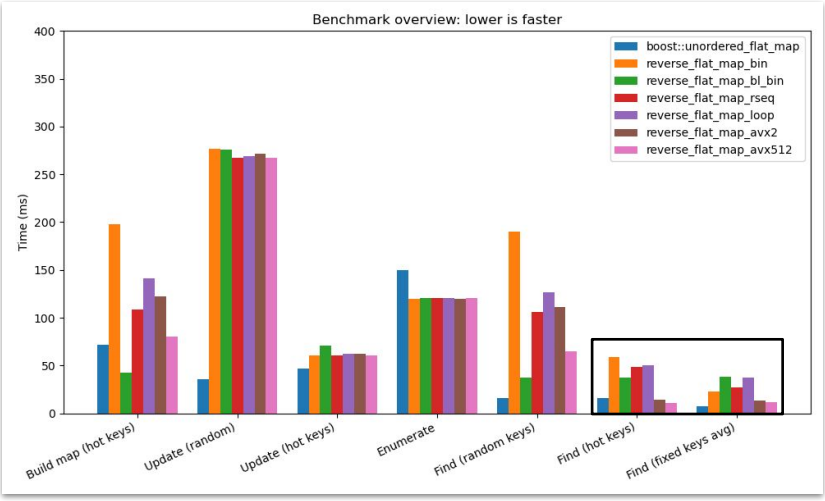
- With **AVX-512**, we load **16 int32 keys at once** from the tail (`_mm512_loadu_si512`), compare them to the **broadcasted key**, and get a **bitmask** of 'greater than'.
- Then we find the first position where the sequence **drops less than or equal to the key** by counting leading zeros in that mask.
- Because most lookups are at the top, the loop almost never needs a second iteration; it returns after the **first 16-key probe**.
- Any remainder (<16) falls back to a tiny scalar tail.

Loads are **unaligned** (we scan from the end), which is fine on modern CPUs; but note that we typically **touch two cache lines**.
We also compile a scalar fallback when AVX-512 isn't available (AVX2/SSE or plain loop).
I have created the AVX2 alternative as well (see the **Godbolt link**).

Benchmark - reverse_flat_map

Qminers

- Various lookup methods
  - type … `array<byte, 1024>`
  - size = 21
- 10M operations
  - Create
  - Insert/Delete
    - Random keys
    - Hot keys
  - Enumerate
  - Lookup
    - Random keys
    - Hot keys
    - Fixed keys



Alright, let's sum up how our flat layout compares to the baseline - the `unordered_flat_map` - which is known to be really fast.

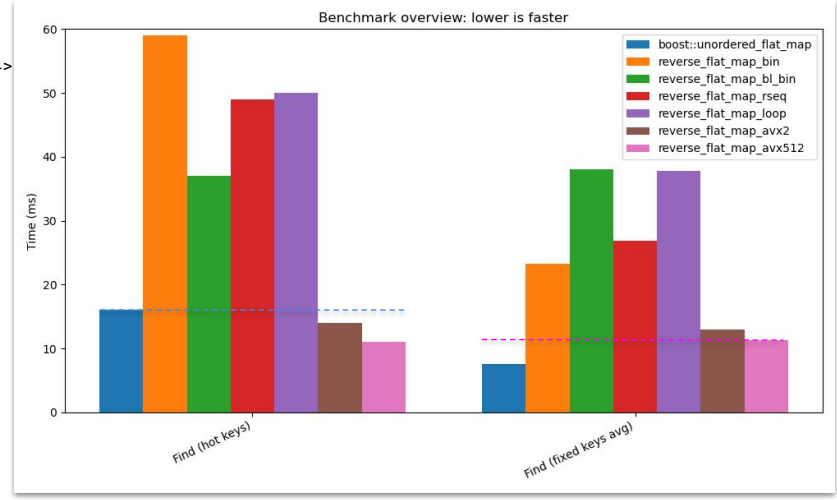
We tested the same small 20-level book with 10 million operations. We looked at build time, lookups, enumeration, and updates.

- **Build time**: `reverse_flat_map` creation is as fast as the `unordered_flat_map`
- **Updates**: For updates at the best level `unordered_flat_map` is fastest, but `reverse_flat_map` is competitive because it's just a quick push or pop (for random keys it is worse due to the shifts).
- **Enumeration**: The flat layout comes out ahead when walking through prices in order - so both `flat_map` and `reverse_flat_map` win.
- **Lookups**: Our design with vectorized scanning wins out, but only for the top-level hot lookups (**for uniform lookups the hashing still has edge**)

In other words, **for a small, top-heavy book, our flat, reverse-ordered design with vectorized lookups gives very nice performance**.

Benchmark - reverse_flat_map

- Various lookup methods
  - type … `array<byte, 1024>`
  - size = 21
- 10M operations
  - Create
  - Insert/Delete
    - Hot keys
    - Random keys
  - Enumerate
  - Lookup
    - Random keys
    - Hot keys
    - Fixed keys



Alright, let's sum up how our flat layout compares to the baseline - the **`unordered_flat_map`** - which is known to be really fast.

We tested the same small 20-level book with 10 million operations. We looked at build time, lookups, enumeration, and updates.

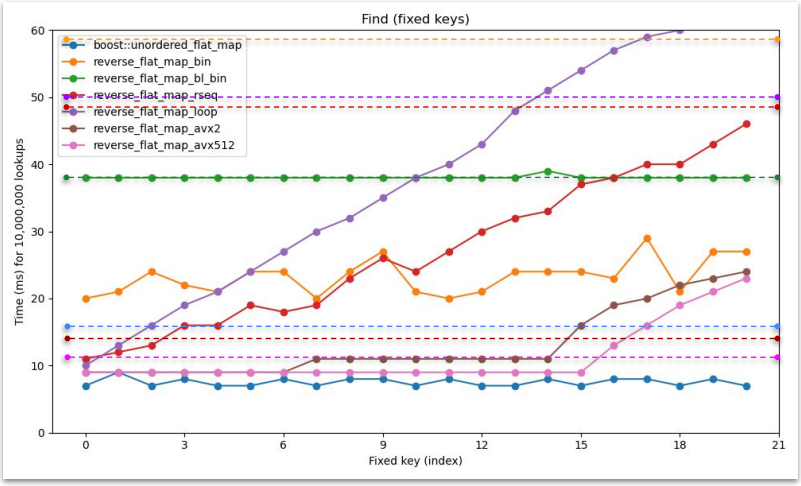
- **Build time**: **`reverse_flat_map`** creation is as fast as the **`unordered_flat_map`**
- **Updates**: For updates at the best level **`unordered_flat_map`** is fastest, but **`reverse_flat_map`** is competitive because it's just a quick push or pop (for random keys it is worse due to the shifts).
- **Enumeration**: The flat layout comes out ahead when walking through prices in order - so both **`flat_map`** and **`reverse_flat_map`** win.
- **Lookups**: Our design with vectorized scanning wins out, but only for the top-level hot lookups (**for uniform lookups the hashing still has edge**)

In other words, for a small, top-heavy book, our flat, reverse-ordered design with vectorized lookups gives very nice performance.

Benchmark - reverse_flat_map

Qminers

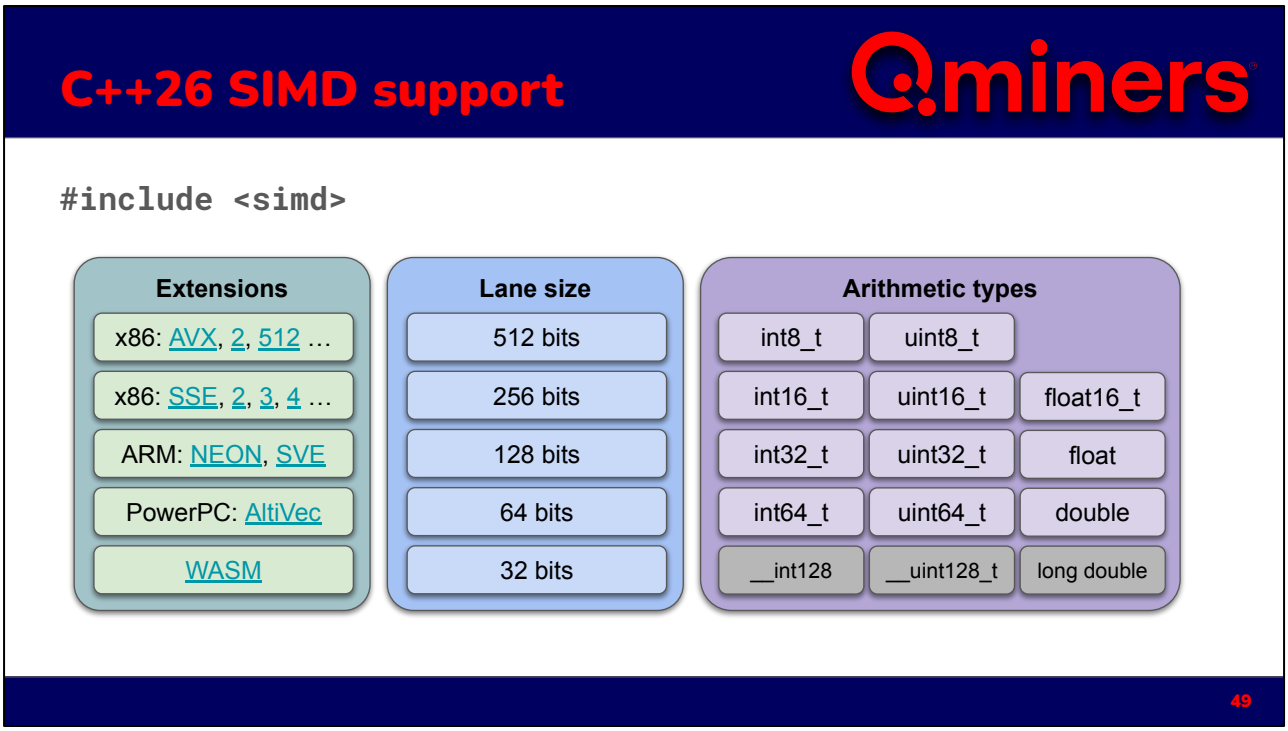
- Various lookup methods
  - type … `array<byte, 1024>`
  - size = 21
- 10M lookups
  - Fixed keys (0 … 20)



So in this test, again we're looking at a best-case scenario where everything is hot in the cache and the branch predictor is fully trained. We're really just **measuring raw instruction speed** here.

- **Binary search** is okay, but its constant overhead is a bit too high for our small table.
  - **Branchless variant of binary search is relatively slow, but note that its performance is extremely stable** (just like the one of hash lookups)
- The **scalar search versions** - one hand-written and one using `std::find_if` - do better. The **hand-written loop generates cleaner assembly**, but the **std version scales a bit better** (due to some tricks like loop unrolling).
- The **vectorized version dominates** and for most keys is **as fast as the O(1) hash lookup**

And let's look at the **hot keys average** - see the dotted lines:
The real winner is the vectorized approach. With AVX-512, we load 16 keys at once and usually find the match in the first probe.
That gives us a steady, super-fast lookup time - about 1.2 nanoseconds on average, compared to around 1.6 for a highly optimized `boost::unordered_flat_map`.
AVX2 alternative is really good too, it just makes a step at key 8 due to second loop iteration (exactly as expected). Both AVX2 and AVX512 then go up linearly since key 16 - also expected for the map of size 21.
One note of caution: on some CPUs, heavy AVX-512 use can cause downclocking, so you should always measure real end-to-end performance and be prepared to resort to narrower lane widths.
Anyway on our target workload, this **vectorized approach is a clear win**.



So the vectorized lookup version is the fastest.
The only downside is that it is **not a fully generic approach**; it's specialized for **int32 keys** - and that's exactly why it's the fastest.
Or is it?
Alright, **here's the exciting news** for anyone who cares about making code run faster: the next version of C++ is going to include something called **standard SIMD**.

Right now, if you want to use those super-fast instructions more generally, you have to write some pretty **complicated code full or preprocessor macros**.
But with C++26, you can write **simple, standard C++ code**, and the compiler will figure out the rest.

It's also **flexible**. You can choose different sizes to match different types of hardware, and it works on different kinds of architectures like x86 and ARM or even Web Assembly.
Basically, you get the **speed of specialized instructions** with the **ease of writing normal code**.

In the end, it's all about getting **faster programs** without having to write complicated, hardware-specific code. It's a win for both performance and simplicity.

Key Lookup SIMD

Qminers

https://godbolt.org/z/bcEMd151x

```
template<typename Key, std::size_t W, typename Tag = simd_ns::element_aligned_tag>
inline bool scan_tail_block(Key key, const Key* keys, std::size_t& n) noexcept {
  using abi_t  = simd_ns::simd_abi::fixed_size<W>;
  using v_t    = simd_ns::simd<Key, abi_t>;
  using mask_t = typename v_t::mask_type;
  const v_t keyv = key;                                  // _mm512_set1_epi32
  for (; n >= W; n -= W) [[likely]] {
    const v_t blk(keys + n - W, Tag{});                  // _mm512_loadu_si512
    const mask_t gt = (blk > keyv);                      // _mm512_cmpgt_epi32_mask
    if (simd_ns::any_of(gt)) [[likely]] {                // mask → bool
        n += simd_ns::find_last_set(gt) + 1 - W;         // std::countl_zero
        return true;
    }
  }
  return false;
}
```


1) Let's look at our vectorized sequential lookup rewritten with c++26 SIMD support:
   You can see the **generic nature** of the C++ **SIMD version in action**.
   With the **same source**, we can instantiate various arithmetic types (both integral and floating point) or lane sizes **16, 8, 4, 2, even 1**.
   And even **combine** them in a cascade - without rewriting the algorithm - for example combine 16 → 8 → 4 widths for most optimal array processing.
2) Now, here are the **three versions of the** gcc 15.2 generated **assembly code** that the compiler generated for AVX, AVX2 and AVX512 respectively.
   It adapts automatically in compile-time: on older CPUs it uses smaller vectors, and on newer ones it uses bigger ones.
   The same code just stretches to fit the hardware.
3) Now we can **highlight the key part of the assembly** - the happy path for our typical lookups.
   It's a short, efficient loop: it loads a chunk of data, compares it, makes a quick decision, and finishes fast.
   Nice compact **15 assembly instructions**, which are packed together, so it is extremely friendly to **instruction cache**.
   We can also look at the **llvm-mca** (machine code analyzer) to see how the **instructions run in parallel** - really efficiently on modern CPUs.

Key Lookup SIMD (cascade)

Qminers

https://godbolt.org/z/bcEMd151x

```
template<typename Key, std::size_t W, typename Tag = simd_ns::element_aligned_tag>
inline bool scan_tail_block(Key key, const Key* keys, std::size_t& n) noexcept {
  using abi_t  = simd_ns::simd_abi::fixed_size<W>;
  using v_t    = simd_ns::simd<Key, ab
  using mask_t = typename v_t::mask_ty
  const v_t keyv = key;
  for (; n >= W; n -= W) [[likely]] {
    const v_t blk(keys + n - W, Tag{})
    const mask_t gt = (blk > keyv);
    if (simd_ns::any_of(gt)) [[likely]]
        n += simd_ns::find_last_set(gt
        return true;
    }
  }
  return false;
}
```

```
template<typename Key,
         std::size_t W = simd_ns::simd<Key, simd_ns::simd_abi::native<Key>>::size()>
inline std::size_t scan_cascade(Key key, const Key* keys, std::size_t n) noexcept {
  if constexpr (W >= 4) {
    if (scan_tail_block<Key, W>(key, keys, n)) [[likely]]
        return n;
    return scan_cascade<Key, W/2>(key, keys, n);
  } else {
    for (; n > 0; --n) {
        if (keys[n-1] > key)
            return n;
    }
    return 0;
  }
}

// Descending lower_bound: first i where keys[i] <= key (assuming descending keys)
template<typename Key>
std::size_t lower_bound_desc_simd(std::span<const Key> keys, Key key) noexcept {
  return scan_cascade(key, keys.data(), keys.size());
}
```

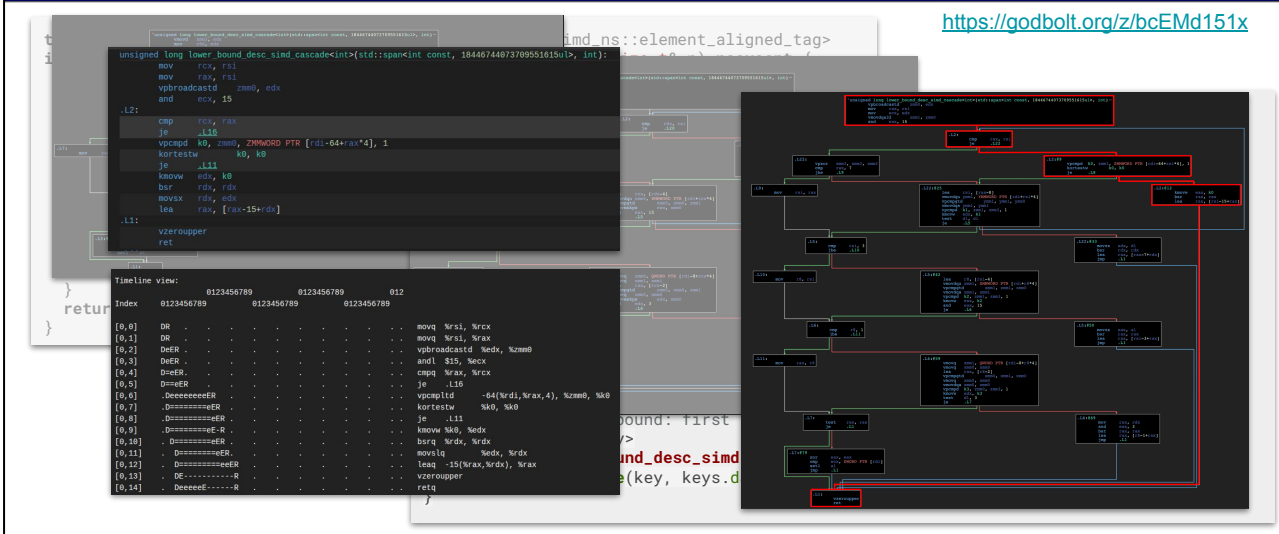

1) Let's look at our vectorized sequential lookup rewritten with c++26 SIMD support:
   You can see the **generic nature** of the C++ **SIMD version in action**.
   With the **same source**, we can instantiate various arithmetic types (both integral and floating point) or lane sizes **16, 8, 4, 2, even 1**.
   And even **combine** them in a cascade - without rewriting the algorithm - for example combine 16 → 8 → 4 widths for most optimal array processing.
2) Now, here are the **three versions of the** gcc 15.2 generated **assembly code** that the compiler generated for AVX, AVX2 and AVX512 respectively.
   It adapts automatically in compile-time: on older CPUs it uses smaller vectors, and on newer ones it uses bigger ones.
   The same code just stretches to fit the hardware.
3) Now we can **highlight the key part of the assembly** - the happy path for our typical lookups.
   It's a short, efficient loop: it loads a chunk of data, compares it, makes a quick decision, and finishes fast.
   Nice compact **15 assembly instructions**, which are packed together, so it is extremely friendly to **instruction cache**.
   We can also look at the **llvm-mca** (machine code analyzer) to see how the **instructions run in parallel** - really efficiently on modern CPUs.

Key Lookup SIMD (lane widths)

Qminers

https://godbolt.org/z/bcEMd151x

1) Let's look at our vectorized sequential lookup rewritten with c++26 SIMD support:
   You can see the **generic nature** of the C++ **SIMD version in action**.
   With the **same source**, we can instantiate various arithmetic types (both integral and floating point) or lane sizes **16, 8, 4, 2, even 1**.
   And even **combine** them in a cascade - without rewriting the algorithm - for example combine 16 → 8 → 4 widths for most optimal array processing.
2) Now, here are the **three versions of the** gcc 15.2 generated **assembly code** that the compiler generated for AVX, AVX2 and AVX512 respectively.
   It adapts automatically in compile-time: on older CPUs it uses smaller vectors, and on newer ones it uses bigger ones.
   The same code just stretches to fit the hardware.
3) Now we can **highlight the key part of the assembly** - the happy path for our typical lookups.
   It's a short, efficient loop: it loads a chunk of data, compares it, makes a quick decision, and finishes fast.
   Nice compact **15 assembly instructions**, which are packed together, so it is extremely friendly to **instruction cache**.
   We can also look at the **llvm-mca** (machine code analyzer) to see how the **instructions run in parallel** - really efficiently on modern CPUs.

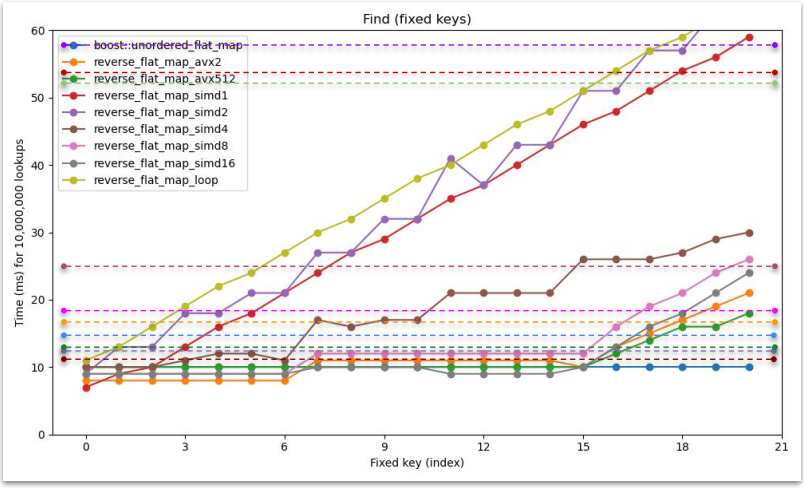


1) Let's look at our vectorized sequential lookup rewritten with c++26 SIMD support:
   You can see the **generic nature** of the C++ **SIMD version in action**.
   With the **same source**, we can instantiate various arithmetic types (both integral and floating point) or lane sizes **16, 8, 4, 2, even 1**.
   And even **combine** them in a cascade - without rewriting the algorithm - for example combine 16 → 8 → 4 widths for most optimal array processing.
2) Now, here are the **three versions of the** gcc 15.2 generated **assembly code** that the compiler generated for AVX, AVX2 and AVX512 respectively.
   It adapts automatically in compile-time: on older CPUs it uses smaller vectors, and on newer ones it uses bigger ones.
   The same code just stretches to fit the hardware.
3) Now we can **highlight the key part of the assembly** - the happy path for our typical lookups.
   It's a short, efficient loop: it loads a chunk of data, compares it, makes a quick decision, and finishes fast.
   Nice compact **15 assembly instructions**, which are packed together, so it is extremely friendly to **instruction cache**.
   We can also look at the **llvm-mca** (machine code analyzer) to see how the **instructions run in parallel** - really efficiently on modern CPUs.

Benchmark - SIMD

Qminers

- Various lookup methods
  - type … `array<byte, 1024>`
  - size = 21
- 10M lookups
  - Fixed keys (0 … 20)
- ~10ms means 1 lookup/ns
  - Throughput ~4 cycles (~4 IPC)
- 15 instructions
  - Latency ~16 cycles



Let's look at the same benchmark again - this time comparing various vectorized versions.
A few highlights:

- **Lane 16/8** (wide vectors) give the best numbers. Just like the non-generic versions.
- **Lane 4** (i.e., **~128-bit** vectors for `int32`) is already **very good** and often close to the wider cases.
- **Lane 1** behaves like a **hand-written scalar loop** - as expected.
- **Lane 2** is a bit **quirky**: the performance isn't perfectly smooth and the gains over scalar are non-existent.
  That's probably normal at such a narrow width - the vector overhead isn't fully amortized, so it can look "noisy."

The key point: it's a **zero-overhead abstraction**. The compiler emits code that's **on par with intrinsics**, so we **don't pay for being generic**.
In short, we get the best of both worlds: **super-fast performance** and **clean, standard code** that works everywhere.

To give a feel for throughput: if we do **10M lookups in ~10 ms**, that's **~1 ns per lookup** (**~1 B lookups/s**).
On a **~4 GHz** CPU (**~4 cycles/ns**), **~15 instructions** per lookup works out to about **4 instructions per cycle (IPC)** in this best-case scenario (cache-hot, no branch misses).

Summary - key takeaways

- **Start with reality - Measure what matters**
  (instrument the system, analyze behavior, build realistic benchmarks)
- **Establish a baseline**
  (pick a strong reference to beat)
- **Pick the right tool for the job**
  (often it is a simple, flat, contiguous layout)
- **Optimize for the hardware**
  (caches & prefetcher, branch predictor, SIMD)
- **Respect constants, not just big-O**
  (*O(N)* can beat *O(logN)* of even *O(1)* ... *in some circumstances)

Start with reality - measure what matters
Don't guess. Instrument the system, **log real-world access patterns**, and **work with actual data**. A **realistic benchmark** shaped by production behavior will beat synthetic microbenchmarks.

Establish a baseline
Before you optimize anything, make sure you have solid, reproducible references. **Pick the strongest version** - challenge yourself.

Pick the right tool for the job
Forget the fancy structures - most of the time, the best tool is simple: a flat array, a tight loop, a **predictable memory layout**.

Optimize for the hardware
Know your **cache lines**. Know your **prefetcher**. Eliminate **unpredictable branches**, use **SIMD** where it makes sense.

Respect constants, not just big-O
Asymptotics are for big N. But **sometimes we are working with small, hot data**. And at that scale, a tight **O(N) loop can outperform O(log N)**, and even **O(1)**.

If we keep doing those things - **choose for locality**, **design for hardware**, and **verify with data** - we'll keep data structure fast where it actually counts.

Links

- Hash maps
  - Boost::unordered documentation
    - https://www.boost.org/doc/libs/1_85_0/libs/unordered/doc/html/unordered.html
  - Joaquín Muñoz - Inside boost::unordered_flat_map
    - https://bannalia.blogspot.com/2022/11/inside-boostunorderedflatmap.html
  - Matt Kulukundis - Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step
    - https://www.youtube.com/watch?v=ncHmEUmJZf4
  - Malte Skarupke - You Can Do Better than std::unordered_map
    - https://www.youtube.com/watch?v=M2fKMP47slQ
- SIMD
  - Matthias Kretz - std::simd
    - https://www.youtube.com/watch?v=LAJ_hywLtMA
  - Denis Yaroshevskiy - Advanced SIMD algorithms in pictures
    - https://www.youtube.com/watch?v=vGcH40rkLdA
- Miscellaneous
  - Andrei Alexandrescu
    - Fastware ... https://www.youtube.com/watch?v=o4-CwDo2zpg
    - Sorting Algorithms: Speed Is Found In The Minds of People ... https://www.youtube.com/watch?v=FJJTYQYB1JQ
    - Rethinking Binary Search: Improving on a Classic with AI Assistance ... https://www.youtube.com/watch?v=FAGf5Xr8HZU
  - Fedor Picus
    - Branchless Programming in C++ ... https://www.youtube.com/watch?v=g-WPhYREFjk
    - Unlocking Modern CPU Power - Next-Gen C++ Optimization Techniques ... https://www.youtube.com/watch?v=wGSSUSeaLgA
  - Bob Steagall ... How to Write a Custom Allocator - https://www.youtube.com/watch?v=kSWfushlvB8
  - Matt Godbolt ... https://www.youtube.com/watch?v=28Gp3TTQYp0
  - David Gross - When Nanoseconds Matter: Ultrafast Trading Systems in C++ ... https://www.youtube.com/watch?v=sX2nF1fW7kI
  - Andreas Weis - Taming Dynamic Memory: An Introduction to Custom Allocators ... https://www.youtube.com/watch?v=lGtKstxNe14
  - Ulrich Drepper - What Every Programmer Should Know About Memory ... https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

Here's a list of resources I've drawn from while putting this presentation together - if you'd like to explore any of the topics deeper, these are great starting points.


Qminers
Thank you
Q&A
57

Perf - focused measurement

Qminers

- Start perf in controlled mode (`--control` parameter)
- Create FIFOs once
- In code (Simple API, RAII)
  - Open the FIFOs at startup
  - Do a warm-up
  - Send "enable\n"
  - Run measured code
  - Send "disable\n"
- Keep runs stable (pin thread, fixed input set, repeat & average)

```cpp
int g_perf_ctl  = -1, g_perf_ack = -1;

void PerfCmd(bool enable) {
  const auto* cmd = enable ? "enable\n" : "disable\n";
  ssize_t n = write(g_perf_ctl, cmd, strlen(cmd));
  std::this_thread::sleep_for(std::chrono::seconds(1));
  char buf[4];
  ssize_t r = read(g_perf_ack, buf, 4);
}

void PerfInit() {
  g_perf_ctl = open("/tmp/perfctl", O_WRONLY|O_CLOEXEC);
  g_perf_ack = open("/tmp/perfack", O_RDONLY|O_CLOEXEC);
  PerfCmd(false);
}

struct PerfScope {
  PerfScope()  { PerfCmd(true); }
  ~PerfScope() { PerfCmd(false); }
};
```

```
$ mkfifo /tmp/perfctl /tmp/perfack
$ perf stat --control="fifo:/tmp/perfctl,/tmp/perfack" ./benchmark
```


First bonus:

Perf can measure exactly the part we care about. I start **`perf stat`** in a special **control mode** that listens on two named pipes. At program start I open those pipes, and when I want to measure I just write **`enable`**; when I'm done, I write **`disable`** and wait for an ack. That lets me **bracket only the inner benchmark** and exclude initialization and cleanup.

I wrap the hot section with a tiny RAII helper so the counters are enabled for the loop and disabled right after. I also do a quick warm-up first so caches and branch predictors are in a steady state. For stability I pin the thread and reuse the same inputs, then repeat a few times and average.

The result is that perf reports **precise numbers for the code that matters** - not the whole process.

Perf (cache-misses) - flat_book


```
$ perf stat --control="fifo:/tmp/perfctl,/tmp/perfack" \
  -e "cycles:u,instructions:u,branches:u,branch-misses:u,cache-references:u,cache-misses:u" ./benchmark
Events disabled
Context: contracts=100, elements=21, lookups=1000000, enumerations=1000000, erases=125000, avgBest=3.75%, avgAny=47.63%
flat_book<int32_t, BigVector>:
  200*2*21 elements creation in 1 ms.
Events enabled
Events disabled
  200*2*1000000 best keys lookups in 1265 ms. (avg=0.79)
Events enabled
Events disabled
  200*2*1000000 any keys lookups in 1295 ms. (avg=10.00)


 Performance counter stats for './benchmark':
     9,507,982,882      cycles:u                                                    (83.34%)
    38,203,289,806      instructions:u                    #    4.02  insn per cycle (83.33%)
     3,975,518,498      branches:u                                                  (83.33%)
         2,791,628      branch-misses:u                   #    0.07% of all branches (83.33%)
     3,701,745,497      cache-references:u                                          (83.33%)
            16,570      cache-misses:u                    #    0.00% of all cache refs (83.34%)

       7.584814391 seconds time elapsed
```


Second bonus:

Earlier we looked at **horizontal traversal** - walking along prices.
This one is the **vertical case**: what happens inside a **hash table** when multiple prices land in the same bucket and we have to walk a **collision chain**.

First, the **baseline**: our `flat_book` doesn't hash at all.
Lookups are a short, sequential scan over a tiny, contiguous array.
Perf shows **almost no cache misses** and very high throughput.

Perf (cache-misses) - hash_book 0.7 Qminers

```
$ perf stat --control="fifo:/tmp/perfctl,/tmp/perfack" \
  -e "cycles:u,instructions:u,branches:u,branch-misses:u,cache-references:u,cache-misses:u" ./benchmark
Events disabled
Context: contracts=100, elements=21, lookups=1000000, enumerations=1000000, erases=125000, avgBest=3.75%, avgAny=47.63%
hash_book<int32_t, BigVector>:
  200*2*21 elements creation in 1 ms.
Events enabled
Events disabled
  200*2*1000000 best keys lookups in 1579 ms. (avg=0.79)
Events enabled
Events disabled
  200*2*1000000 any keys lookups in 1650 ms. (avg=10.00)
LevelBook load factor: 0.6835381235251038

 Performance counter stats for './benchmark':
    12,690,731,875      cycles:u                                                      (83.33%)
    49,625,960,657      instructions:u                    #    3.91  insn per cycle   (83.34%)
     6,406,248,966      branches:u                                                    (83.33%)
         2,709,017      branch-misses:u                   #    0.04% of all branches  (83.33%)
     3,782,805,761      cache-references:u                                            (83.33%)
         4,000,570      cache-misses:u                    #    0.11% of all cache refs (83.33%)

       8.451520926 seconds time elapsed
```


Now the three hash-table runs.
The only thing I change is the **load factor** - think ‘how crowded each bucket is’.

- With a **small load factor** (almost one item per bucket), chains are short and performance is OK.

Perf (cache-misses) - hash_book 5.4

```
$ perf stat --control="fifo:/tmp/perfctl,/tmp/perfack" \
  -e "cycles:u,instructions:u,branches:u,branch-misses:u,cache-references:u,cache-misses:u" ./benchmark
Events disabled
Context: contracts=100, elements=21, lookups=1000000, enumerations=1000000, erases=125000, avgBest=3.75%, avgAny=47.63%
hash_book<int32_t, BigVector>:
  200*2*21 elements creation in 2 ms.
Events enabled
Events disabled
  200*2*1000000 best keys lookups in 1789 ms. (avg=0.79)
Events enabled
Events disabled
  200*2*1000000 any keys lookups in 1775 ms. (avg=10.01)
LevelBook load factor: 5.443940375891121

 Performance counter stats for './benchmark':
    14,627,812,447      cycles:u                                                    (83.33%)
    53,083,627,954      instructions:u                #    3.63  insn per cycle     (83.32%)
     7,176,814,607      branches:u                                                  (83.34%)
         6,282,852      branch-misses:u               #    0.09% of all branches    (83.35%)
     4,247,376,793      cache-references:u                                          (83.34%)
         6,877,731      cache-misses:u                #    0.16% of all cache refs  (83.32%)

       8.985525372 seconds time elapsed
```


- At a **medium load** (~5 per bucket), chains get longer; each step is a **pointer jump** to an address the CPU didn't expect.

Cache misses and time both climb and number of instructions per cycle goes down.

Perf (cache-misses) - hash_book 10.9

```
$ perf stat --control="fifo:/tmp/perfctl,/tmp/perfack" \
  -e "cycles:u,instructions:u,branches:u,branch-misses:u,cache-references:u,cache-misses:u" ./benchmark
Events disabled
Context: contracts=100, elements=21, lookups=1000000, enumerations=1000000, erases=125000, avgBest=3.75%, avgAny=47.63%
hash_book<int32_t, BigVector>:
  200*2*21 elements creation in 2 ms.
Events enabled
Events disabled
  200*2*1000000 best keys lookups in 1887 ms. (avg=0.79)
Events enabled
Events disabled
  200*2*1000000 any keys lookups in 1957 ms. (avg=10.01)
LevelBook load factor: 10.923276983094928

 Performance counter stats for './benchmark':
    15,909,245,429      cycles:u                                                    (83.35%)
    54,711,410,421      instructions:u                    #    3.44  insn per cycle (83.34%)
     7,535,880,012      branches:u                                                  (83.34%)
         5,641,358      branch-misses:u                   #    0.07% of all branches (83.34%)
     5,051,929,212      cache-references:u                                          (83.32%)
        17,730,881      cache-misses:u                    #    0.35% of all cache refs (83.32%)

       9.319218237 seconds time elapsed
```


- With a **high load** (~10 per bucket), we're effectively doing a little **linked-list walk** on almost every lookup - lots of cache misses, lower IPC, and a steep jump in wall-clock time.

Perf (cache-misses) - hash_book

```
$ perf stat --control="fifo:/tmp/perfctl,/tmp/perfack" \
  -e "cycles:u,instructions:u,branches:u,branch-misses:u,ca
Events disabled
Context: contracts=100, elements=21, lookups=1000000, enume
hash_book<int32_t, BigVector>:
  200*2*21 elements creation in 2 ms.
Events enabled
Events disabled
  200*2*1000000 best keys lookups in 1887 ms. (avg=0.79)
Events enabled
Events disabled
  200*2*1000000 any keys lookups in 1957 ms. (avg=10.01)
LevelBook load factor: 10.923276983094928

 Performance counter stats for './benchmark':
    15,909,245,429      cycles:u
    54,711,410,421      instructions:u                #
     7,535,880,012      branches:u
         5,641,358      branch-misses:u               #
     5,051,929,212      cache-references:u
        17,730,881      cache-misses:u                #

       9.319218237 seconds time elapsed
```

```
TDEC_FIXED_UNORDERED_MAP_TEMPLATE_DECL
inline typename TDEC_FIXED_UNORDERED_MAP::const_iterator TDEC_FIXED_UNORDERED_MAP::find(const Key& key) const
{
NodeRef* ptr  = m_buckets + bucket(key);
NodeRef  node = *ptr;
  0.14    mov       0x30(%rdi),%rax
  0.45    mov       (%rax,%rdx,8),%rsi
#ifdef TDEC_FIXED_UNORDERED_MAP_PROFILE_COUNT
unsigned ccount = 0;
#endif
while (node && !Pred()(key, node.key()))
  0.20    test      %rsi,%rsi
  0.41  ↓ jne       1ad
        ↓ jmp       2e8
          nop
Reference next() const { return Reference(Policy::next(m_value)); }
  1a0:    mov       0x18(%rsi),%rsi
while (node && !Pred()(key, node.key()))
  4.66    test      %rsi,%rsi
  0.87  ↓ je        2e8
  1ad:    mov       0x20(%rsi),%rax
 31.51    mov       0x28(%rsi),%rdx
  1.29    xor       %r8,%rax
  3.59    xor       %r9,%rdx
          or        %rdx,%rax
 10.15  ↑ jne       1a0
return level;
}
else
{
typename LevelMap::const_iterator cit = m_levelMap.find(hash);
return (cit != m_levelMap.end()) ? static_cast<Level*>((*cit).second) : 0;
  1c0:    xor       %eax,%eax
  0.44    cmp       0x50(%rdi),%rsi
  0.21    cmove     %rax,%rsi
for (bool isAsk : {false, true}) {
  0.05  1ca:  lea       0x30(%rsp),%rax
const auto val = book.get({cid, isAsk, p})->val;
avg += val;
  0.30    vaddsd    0x30(%rsi),%xmm2,%xmm2
for (bool isAsk : {false, true}) {
  7.57    add       $0x1,%rbx
  0.20    cmp       %rax,%rbx
  0.28  ↑ jne       ea
for (int cid : ctx.contracts) {
          lea       0x4(%r13),%rcx
  0.01    cmp       %rcx,%rbp
  0.07  ↑ jne       d0
          movzwl    0xe(%rsp),%r13d
  0.01    mov       %r12,%rdi
```


The last screenshot is perf's hot-spot view - it lands **exactly** on the instructions that advance through the collision chain.
That's the cost of vertical traversal.

Takeaway: for our small, top-heavy books, a **flat, contiguous layout** avoids these misses entirely.
If a hash table is used, **sizing is critical** - let the load factor creep up, and your theoretical 'O(1) lookup' turns into a cache-miss parade.