Digging Deep for Performance

Mgr. Petr Filipský (MSc)

Table of Contents

Lecture Notes

This document summarizes the key points from the lecture *“Digging Deep for Performance”*, focusing on efficient software development in the field of High Frequency Trading (HFT).

My name is Petr Filipský, and I work at *Qminers*, where I have been part of the team since 2018. Currently, I lead the C++ development team. I graduated in Software Engineering from the Faculty of Mathematics and Physics at Charles University, and the education I received here prepared me well for my professional career. It is a privilege to return to my alma mater and share insights from my work.

Qminers

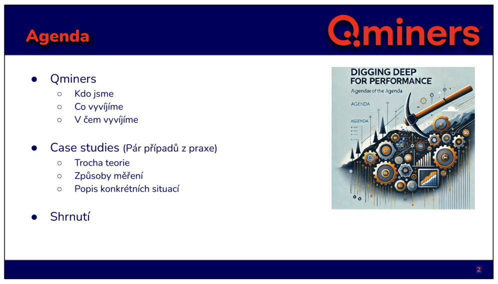
Digging Deep for Performance

Hloubkové dolování výkonu

Mgr. Petr Filipský

About Qminers

- **Foundation and Location** … *Qminers* was founded twelve years ago and is headquartered in the heart of Prague, in the Špork Palace on Hybernská Street.
- **Core Business** … *Qminers* specializes in **High Frequency Trading (HFT)**, a form of algorithmic trading on global financial markets. Our software operates autonomously, relying on mathematical models developed by our analysts.



Agenda

Qminers

- Qminers
  - Kdo jsme
  - Co vyvíjíme
  - V čem vyvíjíme
- Case studies (Pár případů z praxe)
  - Trocha teorie
  - Způsoby měření
  - Popis konkrétních situací
- Shrnutí

2

Technologies Used

- **C++** … The C++ team (7 developers) focuses on building trading software, as well as tools for data preparation and processing.
- **Python** … The Python team (5 developers) creates visualization tools, reporting systems, and analytical solutions that support approximately 30 analysts in their daily tasks

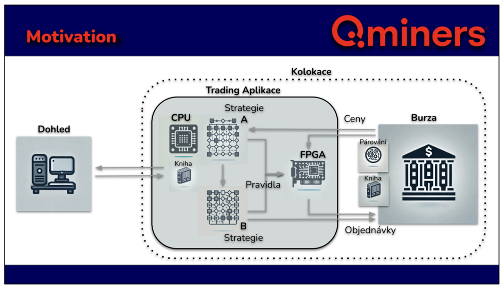
Challenges and Problem Solving

During the lecture, several real-world examples are presented to illustrate the types of challenges tackled by the C++ team. These challenges highlight the critical importance of efficiency and precision in the HFT domain.

While the lecture emphasized issues addressed by the C++ team, it is worth noting that the Python team and analysts face distinct challenges aligned with their specific areas of expertise.

This presentation aims to provide an overview of the work conducted at *Qminers* and the types of problems addressed by its development teams. If you find these topics intriguing, we encourage you to visit us for a deeper understanding of our work.

The Importance of Speed in Trading Systems



This slide illustrates why speed is a critical factor in High Frequency Trading (HFT) and provides a conceptual overview of a modern trading system. On the right, we see the **Exchange**, which acts as a central server responsible for receiving trading requests (e.g., orders), maintaining the **Order Book**, and executing matches via the **Matching Engine**. It continuously shares updates with all participants.

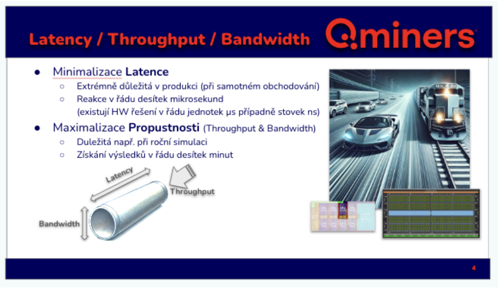
The **Trader Application**, on the other hand, operates as a state machine that processes updates from the exchange and maintains an internal view of the market, including its own copy of the order book. Within this application, **strategies** are implemented to react to market changes as quickly as possible by cancelling, modifying, or placing new orders.

Although the application typically runs within a **colocation facility** (the same data center as the exchange), its internal order book is never perfectly up-to-date due to the asynchronous nature of the system. A fundamental limitation is the **speed of light**, which dictates the maximum speed at which information can propagate (1 nanosecond corresponds to approximately 30 cm). Exchanges mitigate this limitation by ensuring equal cable lengths and similar-quality switches for all participants within the colocation facility, creating a level playing field. However, delays introduced by the trader application itself can vary significantly. Faster implementations lead to a more accurate view of the market, quicker reactions, and a competitive advantage.

To overcome the inherent limitations of CPUs (which typically achieve reaction times in the range of microseconds), parts of the system logic can be offloaded to **dedicated hardware** such as **Field Programmable Gate Arrays (FPGAs)**. These devices operate on nanosecond timescales, allowing for significantly faster reactions. While the trader application functions autonomously, the **workstation** (depicted on the left) serves as a regulatory requirement, enabling operators to monitor trading, adjust strategy parameters, or halt trading entirely. However, due to the autonomous nature of the application, significant financial losses could occur before human intervention is possible in the event of a critical error.

Speed is thus a decisive factor in HFT, where even small delays can result in substantial disadvantages. Optimizing both software performance and hardware utilization is essential to maintaining competitiveness in this highly demanding domain.

Latency vs. Throughput



Latency / Throughput / Bandwidth

- Minimalizace **Latence**
  - Extrémně důležitá v produkci (při samotném obchodování)
  - Reakce v řádu desítek mikrosekund
    (existují HW řešení v řádu jednotek μs případně stovek ns)
- Maximalizace **Propustnosti** (Throughput & Bandwidth)
  - Duležitá např. při roční simulaci
  - Získání výsledků v řádu desítek minut

Qminers
Latency
Throughput
Bandwidth
4

When discussing efficiency in High Frequency Trading, two key aspects are *latency* and *throughput*. These terms represent different dimensions of system performance, each with its own significance depending on the context.

- **Latency** refers to the delay or reaction time between a signal and a response. In trading, low latency is critical in situations where reacting quickly to market events (e.g., a trade affecting future prices) provides a competitive advantage. Faster reactions can directly impact profitability and success.
- **Throughput** represents the number of operations a system can perform within a given time frame. For example, when running multi-year simulations of trading strategies across multiple instruments, high throughput enables faster execution and minimizes cloud computing costs.

A useful analogy contrasts latency and throughput:

- **Latency**: How fast can you travel from Prague to Brno? The focus is on minimizing travel time, so you can use a sports cast or a private jet.
- **Throughput**: How much cargo can be transported from Prague to Brno in a week? A freight train, while slower, is more effective for moving large volumes of goods.

Different hardware is optimized for different tasks, as demonstrated by a comparison of processing power:

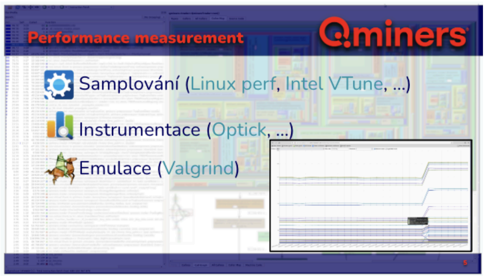
- Intel® Core™ i9-11900 Processor CPU … Peak throughput: ~166 GFLOPS (5.2 GHz × 1 core × 32 instructions per cycle) - Cost: ~$450.
- NVIDIA A100 TENSOR CORE GPU … Peak throughput: ~312 TFLOPS - Cost: >$8000.

Key Takeaways

- **Latency**: Represents how quickly a single task can be completed. For instance, the time it takes to "bring a single child into the world" (analogous to a single-threaded computation). Optimizing latency involves leveraging modern hardware features like superscalar CPUs and SIMD instructions, while respecting physical limits (e.g., speed of light).
- **Throughput**: Reflects the system's ability to process multiple tasks concurrently. For example, the "number of children born in a year" depends on the number of mothers (processors/threads). Increasing throughput involves adding cores, threads, or specialized hardware.

In trading, balancing these two aspects—low latency for real-time decision-making and high throughput for large-scale simulations—is essential for optimal system performance but sometimes goes against each other.

Methods for Measuring Performance


Performance measurement

Qminers

Samplování (Linux perf, Intel VTune, ...)

Instrumentace (Optick, ...)

Emulace (Valgrind)

Performance measurement is a critical aspect of software optimization, particularly in High Frequency Trading. There are several methods used to analyze and improve system performance:

1. **Sampling**
   - At regular intervals, the program's current execution point (call stack) is recorded.
   - Statistically, the frequency of samples for specific code locations is proportional to the time spent there.
2. **Instrumentation** … Specific code sections are instrumented to log timestamps at key points, such as the start and end of a trading strategy's decision-making process.
3. **Emulation** … The program runs on a virtual machine that emulates individual instructions, enabling precise measurement and analysis.

Visualization Tools

The background image on the slide illustrates **KCachegrind**, a visualization tool for analyzing Callgrind output. It provides insights such as *Function lists*, with call counts and instruction counts, *Call maps*, where the area of each block corresponds to the number of instructions executed and *Call graphs*, showing how functions interact.

Another example on the slide depicts a graph tracking the number of instructions executed by various functions over time. This graph is generated daily across all trading products, allowing us to detect performance regressions—unexpected slowdowns in specific parts of the program that require investigation.

Advantages and Limitations of Emulation

Emulation is particularly well-suited for performance monitoring due to its stability. Unlike sampling and instrumentation, emulation results are unaffected by background processes or system load, ensuring consistent and reliable data.

However, emulation also has limitations. The **instruction count**, while useful, is not always a reliable proxy for latency or overall performance. Modern superscalar CPUs can process multiple instructions per cycle at clock speeds of up to 5 GHz. The "cost" of an instruction can vary significantly, from fractions of a nanosecond to ~100 nanoseconds (e.g., in the case of a cache miss requiring access to RAM). This variability spans three orders of magnitude, making instruction counts an imperfect metric for measuring performance comprehensively.

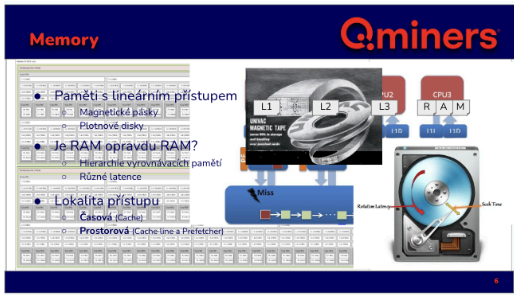
By leveraging the right combination of sampling, instrumentation, and emulation, we can gain deeper insights into performance bottlenecks and address potential regressions effectively.

Memory Access and Its Impact on Performance

Efficient memory access is critical for software performance, as different types of memory exhibit distinct characteristics. Historically, **linear-access memory** supported primarily sequential operations. Examples include **magnetic tapes** (UNIVAC, 1950s), capable of sequentially reading ~7,200 digits per second but requiring up to a minute to rewind, and **hard disk drives (HDDs)**, which achieve ~150 MB/s for sequential reads but suffer from millisecond-level delays during random access due to mechanical repositioning of the read/write head (**seek time**). Optimizations like **Native Command Queuing (NCQ)** reorder requests to improve throughput but can increase latency, reinforcing the need for carefully organized access patterns.

Modern **Random Access Memory (RAM)**, despite its name, does not provide truly uniform access latencies. Instead, the behavior of RAM resembles linear-access memory when considering the CPU's **cache hierarchy**, which acts as an intermediary between the processor and main memory. Access latencies differ substantially across levels of the hierarchy:

- **Level 1 Cache**: ~1 ns latency.
- **Level 2 Cache**: ~7 ns latency.
- **Level 3 Cache**: ~20 ns latency.
- **Main RAM**: ~100 ns latency.



Memory

- Paměti s lineárním přístupem
  - Magnetické pásky
  - Plotnové disky
- Je RAM opravdu RAM?
  - Hierarchie vyrovnávacích pamětí
  - Různé latence
- Lokalita přístupu
  - **Časová** (Cache)
  - **Prostorová** (Cache line a Prefetcher)

Efficient use of this hierarchy relies on adhering to principles of **locality**:

- **Temporal Locality**: Recently accessed addresses are likely to be accessed again soon.
- **Spatial Locality**: Addresses near a recently accessed address are likely to be accessed next.

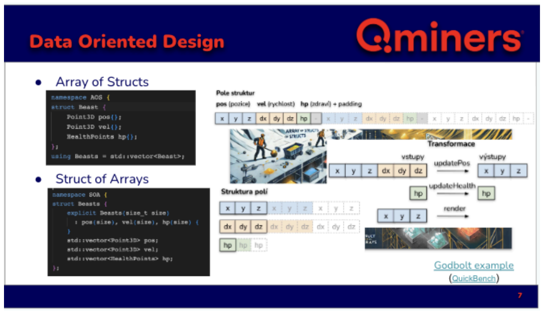
Memory is fetched in **cache lines** (typically 64 bytes), so even single-byte reads load an entire line. Predictable access patterns (e.g., sequential reads) allow the **prefetcher** to preload data into the cache, reducing latency. Poorly structured memory access can disrupt these mechanisms, resulting in cache misses and significant performance penalties. To mitigate this, **data-oriented design** emphasizes structuring data and algorithms to align with hardware behavior.

The importance of these principles is consistent across hardware, despite variations in memory architecture. For example:

1. **NAS (Intel® Celeron® J4025, 2.0 GHz)**: A simpler system with 2 CPUs, 32+24 KB L1 cache, 4 MB L2 cache, and 17 GB RAM.
2. **Development Server (2× AMD EPYC 9454, 3.8 GHz)**: A complex system with 2 CPUs, each with 48 cores and hyper-threading, featuring 48× 32+32 KB L1 cache, 48× 1 MB L2 cache, 8× 32 MB L3 cache, and 2× 756 GB RAM, as well as a NUMA (Non-Uniform Memory Access) architecture.

While these systems differ in cache levels, memory sharing, and NUMA considerations, the **universal principles of efficient memory access—leveraging locality and structuring data to optimize for hardware—apply across all architectures**.

Data-Oriented Design and Memory Efficiency


Data Oriented Design

Qminers

- Array of Structs

```
namespace AOS {
struct Beast {
    Point3D pos{};
    Point3D vel{};
    HealthPoints hp{};
};
using Beasts = std::vector<Beast>;
```

- Struct of Arrays

```
namespace SOA {
struct Beasts {
    explicit Beasts(size_t size)
      : pos(size), vel(size), hp(size) {
    }
    std::vector<Point3D> pos;
    std::vector<Point3D> vel;
    std::vector<HealthPoints> hp;
};
```

Godbolt example
(QuickBench)

7

The hierarchical nature of modern memory systems favors access patterns that align with cache behavior. **Data-Oriented Design** (DOD) is a technique that optimizes software by structuring data to make better use of the memory hierarchy, improving performance and efficiency.

As an example, consider a scenario in game development where you need to represent enemies or creatures. Each creature has attributes such as position, velocity, and health points (**HP**). The typical **Object-Oriented Design** (OOD) approach involves defining a `Beast` structure containing all these attributes and maintaining an **Array of Structs (AoS)**. Operations such as updating positions, modifying health, or rendering would iterate over this array and process each structure.

Comparison of Array of Structs vs. Struct of Arrays

An alternative **Struct of Arrays (SoA)** layout stores each attribute in a separate array. This organization offers significant advantages when not all attributes are required for every operation. By accessing only the relevant arrays, the program avoids unnecessary memory fetches, leveraging the cache more effectively.

Efficiency Gains

- **Position Updates**: Do not require health points
  - Efficiency: *48/56 ≈ 1.167*→ **16.7% improvement**.
- **Health Updates**: Do not require position or velocity.
  - Efficiency: *56/1 = 56* → **56× improvement**.
- **Rendering**: Does not require health or velocity.
  - Efficiency: *56/24 ≈ 2.33* → **2.3× improvement**.

Across all operations, the combined efficiency is:

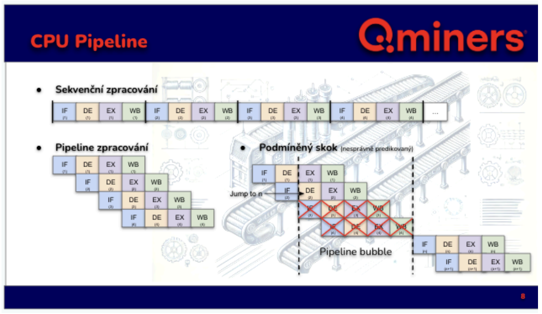
- *3×56 / (48+1+24) ≈ 2.3* → **2.3× improvement**.

Godbolt example: https://godbolt.org/z/Wbd3zjE6c

Conclusion

By restructuring data into a SoA layout, cache usage improves dramatically, allowing more relevant data to fit within a single 64-byte cache line. This leads to measurable performance gains, especially in scenarios where operations only require subsets of attributes. Measuring these improvements in practice demonstrates the potential of **Data-Oriented Design** for creating software that maximizes the capabilities of modern memory architectures.

Instruction Processing in Modern CPUs



Modern CPUs handle machine instructions through a series of stages rather than as atomic operations. These stages—**Fetch**, **Decode**, **Execute**, and **Write Back**—are further broken down into microinstructions.

Early processors, such as the **Intel 80286 (1982)**, processed instructions sequentially. The introduction of a simple three-step pipeline in the **Intel 80386 (1985)** marked the beginning of parallel processing within a single thread. Today's CPUs have evolved into **superscalar architectures**, which allow multiple instructions to be processed simultaneously within a single thread.

Pipeline and Speculative Execution

A **pipeline** enables instructions at different stages to overlap, maximizing throughput. However, this process encounters challenges when instructions involve **conditional jumps**. In such cases, the outcome of the jump condition determines which instruction to execute next, creating potential delays while the CPU waits for the condition to resolve.

To mitigate this, modern CPUs employ **speculative execution**, where instructions following a conditional jump are executed in advance based on a prediction. If the prediction is correct, the pipeline continues seamlessly. However, when the prediction fails, the pipeline must be cleared, and all speculative computations are discarded. This creates a **pipeline bubble**, which can result in a delay of ten or more clock cycles, significantly impacting performance.

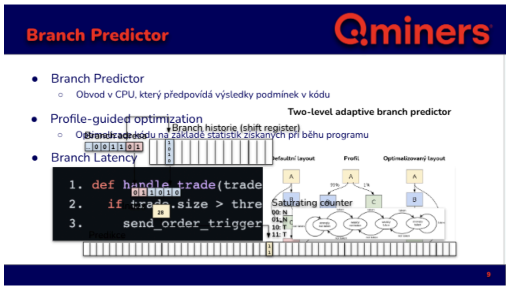
Branch Prediction

The CPU component responsible for these predictions is the **Branch Predictor**, which uses sophisticated algorithms to guess the most likely path of execution. While highly effective in most cases, even minor prediction errors can lead to substantial performance penalties due to pipeline stalls.

By leveraging techniques such as speculative execution and branch prediction, modern CPUs achieve remarkable instruction throughput. However, understanding these mechanisms is essential for optimizing software to minimize pipeline disruptions and maximize performance.

Branch Prediction and Code Optimization

Branch prediction is critical in modern CPUs, enabling speculative execution to prevent pipeline stalls caused by conditional jumps. The **Two-level Adaptive Branch Predictor**, used in Pentium Pro, Pentium II, and Pentium III processors, combines a **4-bit branch history register** with a **Saturating counter**, achieving ~94% prediction accuracy. Incorrect predictions, however, result in pipeline flushes, causing delays of 10–30 cycles (~10 ns) or more.


Branch Predictor

- Branch Predictor
  - Obvod v CPU, který předpovídá výsledky podmínek v kódu
- Profile-guided optimization
  - Optimalizace kódu na základě statistik získaných při běhu programu
- Branch Latency

```
1. def handle_trade(trade
2.   if trade.size > thre
3.     send_order_trigger
```
Qminers
Two-level adaptive branch predictor
Branch historie (shift register)
Branch adresa
Saturating counter
00: N
01: N
10: T
11: T
Predikce
Defaultní layout
Profil
Optimalizovaný layout
99%
1%
A
B
C
28
9

Profile-Guided Optimization (PGO)

PGO enhances branch prediction at the compiler level by using runtime data. Code is first compiled with instrumentation to collect branch outcomes during test runs, generating a profile. This profile is then fed back into the compiler to optimize the machine code based on the most probable branch outcomes. While PGO significantly improves throughput, it can be less effective for latency-sensitive scenarios where rare branches must execute quickly.

Latency-Sensitive Example: `handle_trade`

In a state machine processing trades, a branch evaluates whether a trade exceeds a size threshold to trigger a response. If large trades are rare, minimizing latency for this branch is crucial. Mis-predictions in such cases can incur:

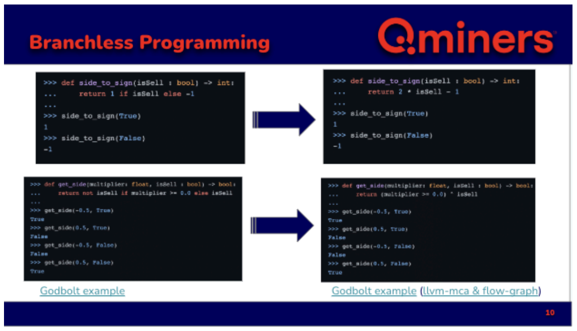
- **Pipeline Stalls**: 10–30 cycles (~10 ns).
- **L3 Cache Misses**: Up to 300 cycles (~100 ns), potentially compounding delays.

Alternatives to Branch Prediction

1. **Data-Driven Logic**: Eliminate branches by processing all trades and tagging smaller ones for later filtering. This avoids prediction penalties but requires significant architectural changes.
2. **Field Programmable Gate Arrays (FPGAs)**: Specialized hardware capable of evaluating conditions and responding in tens of nanoseconds, ideal for ultra-low-latency systems.

Balancing branch prediction, PGO, and advanced hardware solutions is essential for optimizing throughput and latency in performance-critical systems.

Branchless Programming


Branchless Programming

```
>>> def side_to_sign(isSell : bool) -> int:
...     return 1 if isSell else -1
...
>>> side_to_sign(True)
1
>>> side_to_sign(False)
-1
```

```
>>> def side_to_sign(isSell : bool) -> int:
...     return 2 * isSell - 1
...
>>> side_to_sign(True)
1
>>> side_to_sign(False)
-1
```

```
>>> def get_side(multiplier: float, isSell : bool) -> bool:
...     return not isSell if multiplier >= 0.0 else isSell
...
>>> get_side(-0.5, True)
True
>>> get_side(0.5, True)
False
>>> get_side(-0.5, False)
False
>>> get_side(0.5, False)
True
```

```
>>> def get_side(multiplier: float, isSell : bool) -> bool:
...     return (multiplier >= 0.0) ^ isSell
...
>>> get_side(-0.5, True)
True
>>> get_side(0.5, True)
False
>>> get_side(-0.5, False)
False
>>> get_side(0.5, False)
True
```

Godbolt example

Godbolt example (llvm-mca & flow-graph)

10

Branchless programming is a technique that minimizes or eliminates branching in code, often trading a higher instruction count for better pipeline efficiency. By reducing the reliance on conditional jumps, this approach helps avoid branch mis-predictions and their associated performance penalties. Below are examples illustrating branchless implementations and their benefits.

Example 1: `side_to_sign` (Godbolt example: https://godbolt.org/z/f39hf4GP8)

This function determines the sign (1 or -1) based on the side of a trade (e.g., BID or ASK).

- **Branching Version**: Implemented with an `if/else` statement in Python or a ternary operator in C++.
- **Branchless Version**: Uses an arithmetic operation to calculate the result without branching.

Example 2: `get_side` (Godbolt example: https://godbolt.org/z/M761sK3xd)

This function adjusts the sign of the `isSell` argument based on the sign of `multiplier`.

- **Branching Version**: Uses a ternary operator (`? :`) to choose between flipping or retaining the sign.
- **Branchless Version**: Achieves the same result using an **XOR** operation.

The **Godbolt Compiler Explorer** provides a detailed view of the resulting assembly instructions for both approaches. The branchless version typically produces simpler and more predictable instruction sequences, improving performance by avoiding pipeline stalls.

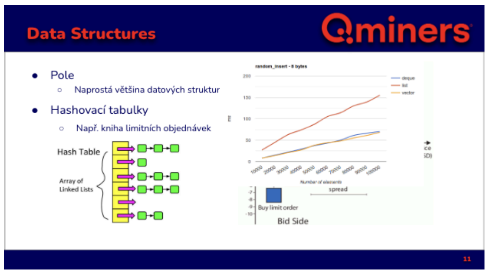
Analyzing Assembly with LLVM Machine Code Analyzer (llvm-mca)

The `llvm-mca` tool simulates instruction execution on a target CPU, highlighting performance characteristics such as dispatch, execution, and retirement stages:

- **D**: Instruction dispatched.
- **e**: Instruction executing.
- **E**: Instruction executed.
- **R**: Instruction retired.

Branchless programming, combined with tools like `llvm-mca` and Godbolt Compiler Explorer, enables developers to write code that better aligns with CPU architecture, optimizing for speed and efficiency while reducing the impact of branching.

Choosing Efficient Data Structures


Data Structures

- Pole
  - Naprostá většina datových struktur
- Hashovací tabulky
  - Např. kniha limitních objednávek

11
Qminers
Hash Table
Array of Linked Lists
random_insert - 8 bytes
deque
list
vector
ms
Number of elements
spread
Buy limit order
Bid Side

For most use cases, **arrays** (or vectors) are the optimal data structure. Their elements are stored in contiguous memory, enabling sequential access, which aligns well with modern hardware and cache hierarchies.

Insertion Speed: Vector vs. List

The graph demonstrates the speed of inserting elements at random positions in three data structures: vectors, deques, and lists. While theory suggests that linked lists (with O(1) insertion) should outperform vectors (with O(N) insertion), the results show the opposite. Despite the overhead of shifting elements, the sequential memory layout of vectors allows hardware to process these operations efficiently, outperforming lists, where elements are scattered in memory.

Hash Tables and Collision Management

Hash tables are another frequently used data structure. A typical hash table consists of an array of buckets indexed by a hash of the key. When collisions occur (two keys hashing to the same index), a **collision chain** is formed, often as a linked list. As the hash table fills, these chains grow, and performance can degrade from O(1) to O(N), resembling the drawbacks of linked lists.

The need to dimension hash tables appropriately becomes critical in trading applications. For example, our order book software operates across many markets, each with unique characteristics:

- **Tick Size**: Defines the granularity of the price grid (e.g., integer-only prices for a tick size of one).
- **Price Behavior**: Some markets exhibit stable price oscillations, while others are highly volatile.

Order Book Visualization

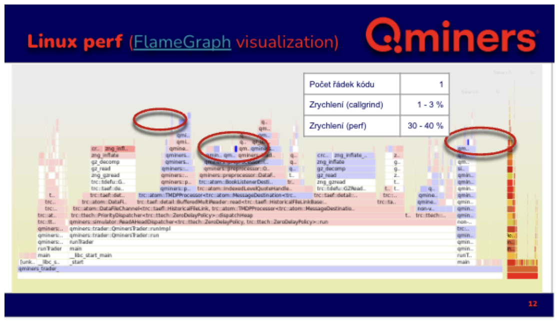
The accompanying image shows an **order book** for a single contract, where the X-axis represents price and the Y-axis shows the quantity of lots participants want to buy (blue) or sell (red). Each market and commodity has its own order book, influenced by factors like tick size and volatility, requiring careful adjustment of underlying data structures to optimize performance.

Efficiently selecting and tuning data structures like vectors and hash tables is key to achieving high performance in trading applications, where the balance between theoretical complexity and real-world hardware efficiency often determines success.

Improperly sized hash tables can lead to long collision chains, undermining performance. By tailoring hash table parameters to market-specific traits, such as volatility and tick size, these issues can be mitigated.

Case Study: Hash Table Dimensioning and Performance Bottlenecks

A significant performance issue arose in our system after introducing a new market with a fine price grid and high price volatility. The hash table, used to store order book data, was undersized, resulting in excessively long collision chains. This caused operations on the table to exhibit performance characteristics similar to sequential searches in a linked list. Each traversal between entries in the collision chain (illustrated by green squares) typically triggered a **cache miss** and sometimes a **branch misprediction**, making lookups extremely costly.



Počet řádek kódu	1
Zrychlení (callgrind)	1 - 3 %
Zrychlení (perf)	30 - 40 %

Why the Problem Wasn't Immediately Apparent

Despite monitoring performance metrics, the issue was not immediately detected because the number of instructions executed did not increase significantly. While instruction count is often a useful metric, it can be misleading in cases like this, where time spent per operation is a more relevant measure.

Using Flame Graphs to Diagnose the Issue

The problem was identified through flame graph analysis, which visualizes performance sampling data based on actual time spent in different code sections.

- **Initial Graph**: Highlighted bottlenecks in two areas of the trading strategy and one in the simulator, all related to hash table lookups.
- **Optimized Graph**: After resizing the hash table to reduce collision chains, the bottlenecks were eliminated.

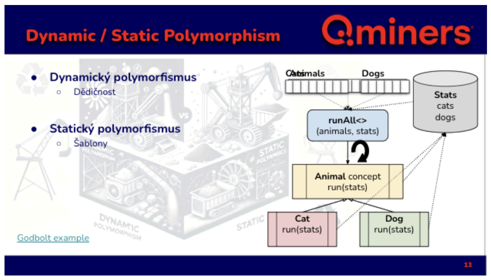
A **diff between the graphs** revealed striking differences:

- **Instruction Count**: Minor differences, only a few percent.
- **Execution Time**: Differences of tens of percent, demonstrating the true performance impact.

Lessons Learned

This case underscores the limitations of relying solely on instruction count as a performance metric. When dealing with performance-critical systems, especially those involving memory-bound operations, measuring actual execution time is essential. Addressing this issue improved performance significantly and reinforced the importance of carefully sizing data structures to match workload characteristics.

Dynamic vs. Static Polymorphism



Dynamic / Static Polymorphism

- **Dynamický polymorfismus**
  - Dědičnost
- **Statický polymorfismus**
  - Šablony

Godbolt example

13

Qminers
Cats
Animals
Dogs
Stats
cats
dogs
runAll<>
(animals, stats)
Animal concept
run(stats)
Cat
run(stats)
Dog
run(stats)
DYNAMIC POLYMORPHISM
STATIC

Polymorphism is a core concept in programming, enabling flexibility and extensibility. Here, we compare **dynamic** and **static** polymorphism, focusing on their performance implications and use cases.

Dynamic Polymorphism

Dynamic polymorphism is common in object-oriented programming. It involves an interface, like `IAnimal`, and multiple implementations, such as `Dog` and `Cat`. In this example:

- A `Stats` structure tracks counts of dogs and cats.
- All animals are stored in a single array, and the `runAll` function iterates through this array, invoking the `run` method for each animal.
- Each animal increments the appropriate counter in `Stats`.

However, calling `run` is relatively expensive due to **type erasure**, where type-specific information is intentionally discarded. This results in a **branch** (difficult for the branch predictor to handle) or an **indirection** via a virtual method table (potentially causing instruction cache misses). In either case, the runtime cannot know in advance which version of `run` will be invoked, leading to performance overhead.

Static Polymorphism

Static polymorphism, often achieved using **templates** in C++, eliminates this overhead by retaining full type information at compile time. For the same example: Instead of using an interface, `Animal` is a **concept**, implemented by `Cat` and `Dog`. Cats and dogs are stored in separate arrays, and `runAll` becomes a template function.

This approach allows the compiler to generate type-specific code, avoiding type erasure and performing aggressive optimizations, as all type information is available at compile time - like complete loop elimination.

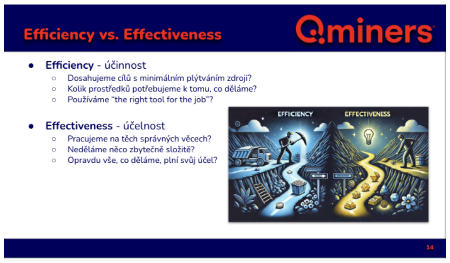
Performance Comparison (Godbolt example: https://godbolt.org/z/qenjv9PaM)

Using the **Godbolt Compiler Explorer**, we can observe the assembly code for both approaches. The dynamic version involves indirect calls or branches, while the static version generates straightforward, predictable code, reducing overhead and improving execution speed.

Key Takeaways

Dynamic polymorphism offers flexibility but incurs runtime costs due to type erasure and indirect calls. Static polymorphism, where applicable, avoids these issues and allows for optimized, type-specific code generation. Choosing the right approach depends on the balance between runtime flexibility and performance needs.

Efficiency vs. Effectiveness



Efficiency vs. Effectiveness

Qminers

- **Efficiency** - účinnost
  - Dosahujeme cílů s minimálním plýtváním zdroji?
  - Kolik prostředků potřebujeme k tomu, co děláme?
  - Používáme "the right tool for the job"?
- **Effectiveness** - účelnost
  - Pracujeme na těch správných věcech?
  - Neděláme něco zbytečně složitě?
  - Opravdu vše, co děláme, plní svůj účel?

EFFICIENCY
EFFECTIVENESS

14

In High Frequency Trading, speed and efficiency are critical. Significant effort is invested into ensuring that software not only performs correctly but also achieves maximum efficiency. This dual focus makes the field both challenging and rewarding.

Efficiency: Implementing Solutions Optimally

Once a requirement is identified, it must be implemented as efficiently as possible under the given constraints. Efficiency involves designing solutions that minimize computational costs while maximizing performance.

In trading, efficiency is not just a secondary consideration but a core necessity. Even a correct and otherwise effective solution can fail if it is not efficient enough to function in real-world conditions.

Effectiveness: Choosing the Right Goals

Before efficiency can even come into play, it is vital to ensure that the selected goals align with the broader objectives of the system. The team must navigate a constant influx of feature requests and requirements, carefully filtering to focus on those that genuinely advance our goals.

This is far from straightforward, as the goals themselves are often dynamic and not clearly defined. Achieving effectiveness requires a deep understanding of the system's context and the foresight to prioritize the most impactful requests.

Key Takeaways

Trading is one of the few fields where both efficiency and effectiveness are critical to success:

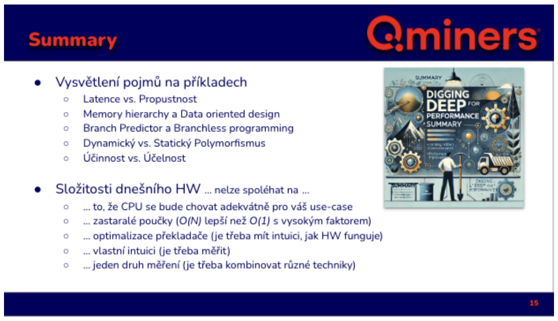
1. **Efficiency** ensures optimal use of resources, allowing the solution to operate in real-time markets.
2. **Effectiveness** ensures that efforts are directed toward meaningful objectives that align with the system's purpose.

This dual focus guarantees that software is not only functional and correct but also impactful and competitive in demanding trading environments.

Summary

As we approach the end of this lecture (excluding the bonus slides for further exploration), let's recap the key topics we've covered:

- *Latency* vs. *Throughput*
- *Memory Hierarchy* and *Data-Oriented Design*
- *Branch Predictor* and *Branchless Programming*
- *Dynamic* vs. *Static Polymorphism*
- *Efficiency* vs. *Effectiveness*



Summary

- Vysvětlení pojmů na příkladech
  - Latence vs. Propustnost
  - Memory hierarchy a Data oriented design
  - Branch Predictor a Branchless programming
  - Dynamický vs. Statický Polymorfismus
  - Účinnost vs. Účelnost
- Složitosti dnešního HW ... nelze spoléhat na ...
  - ... to, že CPU se bude chovat adekvátně pro váš use-case
  - ... zastaralé poučky (*O(N)* lepší než *O(1)* s vysokým faktorem)
  - ... optimalizace překladače (je třeba mít intuici, jak HW funguje)
  - ... vlastní intuici (je třeba měřit)
  - ... jeden druh měření (je třeba kombinovat různé techniky)

15

Key Insights

Today's hardware is incredibly complex, and professional software development must adapt to these intricacies. To build efficient and effective systems, several principles must be followed:

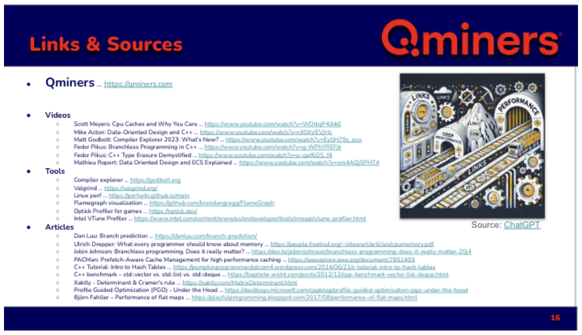
1. **Challenge Outdated Assumptions**:
   - Rules that once applied (e.g., linked lists vs. vectors) may no longer hold due to advances in hardware.
2. **Understand the Hardware**:
   - Compilers are powerful, but they can't handle everything. Developers need to cultivate an intuition for how modern hardware behaves.
3. **Measure and Verify**:
   - Intuition alone is insufficient. Measurements often reveal counterintuitive results or the scale of a performance gap that intuition underestimated.
4. **Use Diverse Metrics**:
   - Relying on a single metric (e.g., instruction count) can be misleading. Combining different types of measurements provides a more accurate picture of performance.

Adapting to modern hardware requires not only technical knowledge but also critical thinking, empirical validation, and the ability to challenge preconceived notions. This combination ensures software that is both performant and reliable in today's demanding environments.

Resources

The following resources were utilized during the preparation of this lecture:

- **Videos**:
  A selection of lectures and presentations offering insights into the topics discussed.
  - Scott Meyers: Cpu Caches and Why You Care
  - Mike Acton: Data-Oriented Design and C++
  - Matt Godbolt: Compiler Explorer 2023: What's New?
  - Fedor Pikus: Branchless Programming in C++
  - Fedor Pikus: C++ Type Erasure Demystified
  - Mathieu Ropert: Data Oriented Design and ECS Explained
- **Tools**:
  Links to performance analysis tools that facilitate identifying and addressing bottlenecks
  - Godbolt Compiler Explorer
  - Valgrind
  - Linux perf
  - Flamegraph visualization
  - Optick Profiler for games
  - Intel VTune Profiler



Links & Sources

- Qminers ... https://qminers.com
- Videos
  - Scott Meyers: Cpu Caches and Why You Care ... https://www.youtube.com/watch?v=WDIkqP4JbkE
  - Mike Acton: Data-Oriented Design and C++ ... https://www.youtube.com/watch?v=rX0ItVEVjHc
  - Matt Godbolt: Compiler Explorer 2023: What's New? ... https://www.youtube.com/watch?v=Ey0H79z_pco
  - Fedor Pikus: Branchless Programming in C++ ... https://www.youtube.com/watch?v=g-WPhYREFjk
  - Fedor Pikus: C++ Type Erasure Demystified ... https://www.youtube.com/watch?v=p-qaf6OS_f4
  - Mathieu Ropert: Data Oriented Design and ECS Explained ... https://www.youtube.com/watch?v=xm4AQj5PHT4
- Tools
  - Compiler explorer ... https://godbolt.org
  - Valgrind ... https://valgrind.org/
  - Linux perf ... https://perfwiki.github.io/main
  - Flamegraph visualization ... https://github.com/brendangregg/FlameGraph
  - Optick Profiler for games ... https://optick.dev/
  - Intel VTune Profiler ... https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
- Articles
  - Dan Luu: Branch prediction ... https://danluu.com/branch-prediction/
  - Ulrich Drepper: What every programmer should know about memory ... https://people.freebsd.org/~lstewart/articles/cpumemory.pdf
  - Jobin Johnson: Branchless programming. Does it really matter? ... https://dev.to/jobinrjohnson/branchless-programming-does-it-really-matter-20j4
  - PACMan: Prefetch-Aware Cache Management for high performance caching ... https://ieeexplore.ieee.org/document/7851493
  - C++ Tutorial: Intro to Hash Tables ... https://pumpkinprogrammerdotcom4.wordpress.com/2014/06/21/c-tutorial-intro-to-hash-tables
  - C++ benchmark – std::vector vs. std::list vs. std::deque ... https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html
  - Xaktly - Determinant & Cramer's rule ... https://xaktly.com/MatrixDeterminant.html
  - Profile Guided Optimization (PGO) – Under the Hood ... https://devblogs.microsoft.com/cppblog/profile-guided-optimization-pgo-under-the-hood
  - Björn Fahller – Performance of flat maps ... https://playfulprogramming.blogspot.com/2017/08/performance-of-flat-maps.html

Source: ChatGPT

16

- **Articles**:
  Detailed explorations of the concepts covered in this lecture, providing greater depth than time allowed here.
  - Dan Luu: Branch prediction
  - Ulrich Drepper: What every programmer should know about memory
  - Jobin Johnson: Branchless programming. Does it really matter?
  - PACMan: Prefetch-Aware Cache Management for high performance caching
  - C++ Tutorial: Intro to Hash Tables
  - C++ benchmark – std::vector vs. std::list vs. std::deque
  - Xaktly - Determinant & Cramer's rule
  - Profile Guided Optimization (PGO) – Under the Hood
  - Björn Fahller – Performance of flat maps

Conclusion

Děkuji za pozornost

17

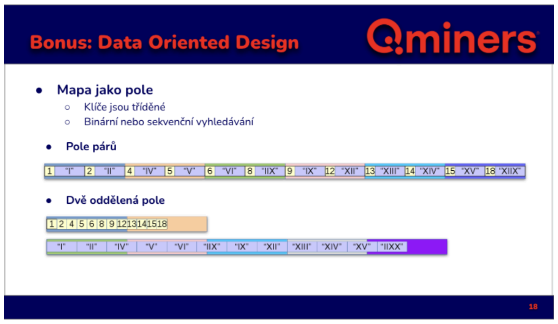
This concludes the main part of the lecture. The goal was to provide a comprehensive overview of key topics in software performance and their relationship to modern hardware. It is hoped that the content has sparked interest and provided valuable insights into this field.

For those interested in learning more about these concepts or the work we do at Qminers, further engagement is encouraged.

If time allows, questions are welcome. Alternatively, the bonus slides are available for a deeper dive into some of the topics introduced earlier.

Bonus Material

Flat Map and Cache Efficiency



Bonus: Data Oriented Design

Qminers

- **Mapa jako pole**
  - Klíče jsou tříděné
  - Binární nebo sekvenční vyhledávání
- **Pole párů**

1 "I" 2 "II" 4 "IV" 5 "V" 6 "VI" 8 "IIX" 9 "IX" 12 "XII" 13 "XIII" 14 "XIV" 15 "XV" 18 "XIIX"

- **Dvě oddělená pole**

1 2 4 5 6 8 9 12 13 14 15 18

"I" "II" "IV" "V" "VI" "IIX" "IX" "XII" "XIII" "XIV" "XV" "IIXX"

18

Building on the earlier discussion of **Data-Oriented Design** and the comparison of Array of Structs (AoS) vs. Struct of Arrays (SoA), a similar principle applies to the **flat_map**—an associative container that maps keys to values.

Structure of `flat_map`

In a `flat_map`, keys and values are stored in **separate arrays**, as opposed to traditional maps where each key-value pair is stored together. This separation offers significant advantages in terms of cache efficiency:

- The **key array** makes more efficient use of cache lines, fitting more keys into a single cache line.
- During lookups, only the key array is traversed, and values are accessed only after the correct key is found.

Cache Behavior

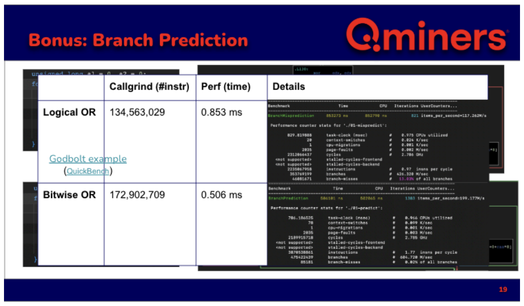
This design reduces unnecessary memory access:

1. **Key Lookup**: The traversal of the key array efficiently utilizes the CPU cache, as multiple keys can fit in a single cache line.
2. **Value Access**: Once the correct key is located, only a single **data cache miss** is incurred to access the corresponding value in the value array.

By minimizing random memory access, `flat_map` aligns well with modern hardware, leveraging its cache hierarchies to achieve improved performance compared to traditional associative containers.

Branch Predictor: Advanced Considerations

Branch prediction is a critical mechanism in modern CPUs, but its effectiveness depends heavily on the predictability of the conditions being evaluated. A compelling example of this challenge was presented by Fedor G. Picus in his lecture *Branchless Programming in C++*.



Bonus: Branch Prediction

	Callgrind (#instr)	Perf (time)	Details
Logical OR Godbolt example (QuickBench)	134,563,029	0.853 ms	
Bitwise OR	172,902,709	0.506 ms	

```
Benchmark                 Time            CPU   Iterations UserCounters...
BranchMisprediction  853273 ns      852790 ns          821 items_per_second=117.262M/s

 Performance counter stats for './01-mispredict':

        829.819888      task-clock (msec)         #    0.975 CPUs utilized
                20      context-switches          #    0.024 K/sec
                 1      cpu-migrations            #    0.001 K/sec
              2035      page-faults               #    0.002 M/sec
        2312066437      cycles                    #    2.786 GHz
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
        2235067958      instructions              #    0.97  insns per cycle
         353769199      branches                  #  426.320 M/sec
          46081671      branch-misses             #   13.03% of all branches
```

```
Benchmark                 Time            CPU   Iterations UserCounters...
BranchPrediction     506101 ns      502065 ns         1383 items_per_second=199.177M/s

 Performance counter stats for './01-predict':

        786.186525      task-clock (msec)         #    0.966 CPUs utilized
                78      context-switches          #    0.099 K/sec
                 1      cpu-migrations            #    0.001 K/sec
              2035      page-faults               #    0.003 M/sec
        2189915718      cycles                    #    2.785 GHz
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
        3870538861      instructions              #    1.77  insns per cycle
         475422439      branches                  #  604.720 M/sec
             85181      branch-misses             #    0.02% of all branches
```

19

Example: Dual Conditions in a Loop

Consider a loop that evaluates two conditions (B1 and B2) stored in separate vectors. Based on the result of this disjunction (`B1 || B2`), one of two operations is executed.

- **Short-Circuit Evaluation**: In logical disjunctions, the second condition is evaluated only if the first is false. This creates two potential branches, as shown in the **Control Flow Graph (CFG)**.
- **Alternative Approach**: If short-circuit evaluation is unnecessary, the condition can be rewritten using bitwise operations, reducing the two branches to one.

The Challenge with Random Conditions

While it may seem that branch prediction would mitigate the cost of these conditions, randomness in the input can significantly degrade its performance:

- Imagine B1 and B2 being filled with random values, where each condition independently alternates between true and false. Despite the disjunction always evaluating to true, the randomness of B1 and B2 makes each individual condition difficult for the Branch Predictor to predict. From the CPU's perspective, the two conditions are independent, compounding the difficulty.

Optimization Using Bitwise Operations

Replacing the disjunction with a bitwise operation simplifies the control flow:

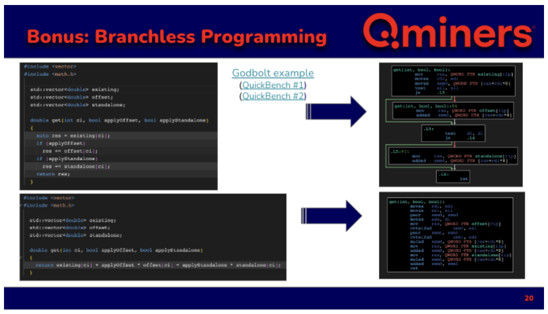
1. Evaluate both B1 and B2 for every iteration using bitwise OR.
2. Perform a single branch based on the result of the bitwise operation.

This optimization trades additional computation for improved predictability. Although more work is done (both conditions are always evaluated), the CPU's **super-scalar architecture** can handle this efficiently, leveraging its available computational resources.

Conclusion (Godbolt example: https://godbolt.org/z/z56bh8qWW)

Random conditions are particularly challenging for Branch Predictors because they lack discernible patterns. By reducing the number of branches and opting for more predictable control flows, performance can be improved, especially in cases where predictability outweighs the cost of additional computation. This optimization, however, should only be applied selectively, as it increases workload in exchange for more consistent branching behavior.

Interdependencies in Performance Optimization



Bonus: Branchless Programming

Qminers

```cpp
#include <vector>
#include <math.h>

std::vector<double> existing;
std::vector<double> offset;
std::vector<double> standalone;

double get(int ci, bool applyOffset, bool applyStandalone)
{
  auto res = existing[ci];
  if (applyOffset)
    res += offset[ci];
  if (applyStandalone)
    res += standalone[ci];
  return res;
}
```

Godbolt example
(QuickBench #1)
(QuickBench #2)

```
get(int, bool, bool):
        mov     rax, QWORD PTR existing[rip]
        movzx   rdi, edi
        movsd   xmm0, QWORD PTR [rax+rdi*8]
        test    sil, sil
        je      .L5
get(int, bool, bool):@6
        mov     rax, QWORD PTR offset[rip]
        addsd   xmm0, QWORD PTR [rax+rdi*8]
.L5:
        test    dl, dl
        je      .L4
.L5:@11
        mov     rax, QWORD PTR standalone[rip]
        addsd   xmm0, QWORD PTR [rax+rdi*8]
.L4:
        ret
```

```cpp
#include <vector>
#include <math.h>

std::vector<double> existing;
std::vector<double> offset;
std::vector<double> standalone;

double get(int ci, bool applyOffset, bool applyStandalone)
{
  return existing[ci] + applyOffset * offset[ci] + applyStandalone * standalone[ci];
}
```

```
get(int, bool, bool):
        movzx   rdi, edi
        movzx   esi, sil
        pxor    xmm0, xmm0
        movzx   edx, dl
        mov     rax, QWORD PTR offset[rip]
        cvtsi2sd        xmm0, esi
        pxor    xmm1, xmm1
        cvtsi2sd        xmm1, edx
        mulsd   xmm0, QWORD PTR [rax+rdi*8]
        mov     rax, QWORD PTR existing[rip]
        addsd   xmm0, QWORD PTR [rax+rdi*8]
        mov     rax, QWORD PTR standalone[rip]
        mulsd   xmm1, QWORD PTR [rax+rdi*8]
        addsd   xmm0, xmm1
        ret
```

20

Techniques like branchless programming are not universally applicable and must be evaluated in context. Several factors influence their effectiveness, including:

- **Branch Predictability**: Well-predicted branches are inexpensive, while poorly predicted branches can justify branchless transformations.
- **Instruction Count**: Branchless code often executes more instructions. While modern superscalar CPUs can handle this efficiently, additional instructions that access uncached memory can significantly degrade performance.

Examples: Branchless vs. Branchful Code

Two extreme scenarios illustrate how context affects performance:

1. **Branchless Code is Faster**:
   - Arrays (`existing`, `offset`, and `standalone`) are traversed sequentially, leveraging cache locality.
   - Conditions (`applyOffset` and `applyStandalone`) are poorly predicted, making branchless code preferable.
2. **Branchless Code is Slower**:
   - Arrays are accessed randomly, resulting in frequent cache misses.
   - Conditions are always false, making them well-predicted and requiring no additional work.

Impact of Transformations (Godbolt example: https://godbolt.org/z/G183q3jKW)

Aggregating the three arrays (`existing`, `offset`, and `standalone`) into a single array reveals further insights:

- **BranchfulSequential**: Performance remains stable, likely CPU-bound due to mispredicted branches.
- **BranchlessSequential**: Performance decreases slightly (14M to 19M iterations/sec), possibly due to more complex offset calculations.
- **BranchfulRandom**: Performance worsens significantly (22M to 55M iterations/sec), as cache efficiency drops when accessing one-third of the values (*Struct of Arrays* vs. *Array of Structs*).
- **BranchlessRandom**: Performance improves (128M to 76M iterations/sec), thanks to better cache locality achieved by the transformation from *Struct of Arrays* to *Array of Structs*.

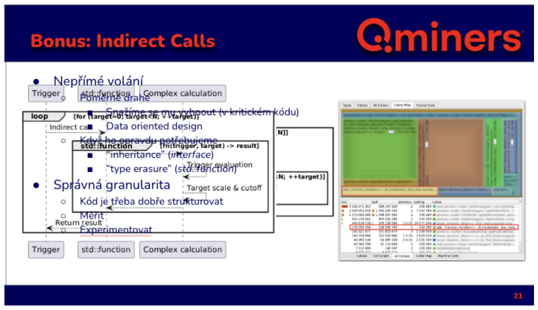
Key Insight

These results demonstrate that optimization rules cannot be applied in isolation—each decision interacts with others, and "everything is interconnected." The optimal approach depends on understanding the entire system, from data structures to memory access patterns and CPU behavior.

Static vs. Dynamic Polymorphism: Revisiting the Trade-offs

As previously discussed, **indirect calls** introduce significant overhead, including:

- **Branch mispredictions** and **instruction cache misses**, which disrupt pipeline efficiency.
- **Reduced compiler optimizations**, as the specific implementation is unknown at compile time.


Bonus: Indirect Calls

- Nepřímé volání
  - Poměrně drahé
    - Snažíme se mu vyhnout (v kritickém kódu)
    - Data oriented design
  - Když ho opravdu potřebujeme
    - "inheritance" (interface)
    - "type erasure" (std::function)
- Správná granularita
  - Kód je třeba dobře strukturovat
  - Měřit
  - Experimentovat

21
Trigger
std::function
Complex calculation
loop
[for (target=0; target<N; ++target)]
Indirect call
std::function
[fn(trigger, target) -> result]
Trigger evaluation
Target scale & cutoff
Return result
N]]
:N; ++target)]
Types
Callers
All Callers
Callee Map
Source Code
Callees
Call Graph
All Callees
Caller Map
Machine Code

Avoiding Indirect Calls

In performance-critical code, avoiding indirect calls is often preferable. This is typically achieved by replacing **dynamic polymorphism** with **static polymorphism** (e.g., templates in C++). However, there are cases where dynamic calls are necessary or practical, such as when using:

- **Inheritance** (e.g., an interface with virtual functions).
- **Type erasure** (e.g., `std::function` or similar abstractions).

When indirect calls cannot be avoided, it is critical to:

1. **Use Appropriate Granularity**: Structure code to minimize the frequency of indirect calls by carefully placing the boundaries where they occur.
2. **Measure and Experiment**: Modern CPU architectures are complex, and behavior is often unintuitive. Testing and refining code, ideally backed by unit tests, is essential.

Practical Example: Loop Placement and Performance

A real-world example highlights the impact of indirect call placement:

- Initial Design: A **hot loop** repeatedly invoked an `std::function`, resulting in up to twenty indirect calls per iteration.
- Optimization: The loop was moved **inside** the `std::function`, consolidating the twenty calls into a single one.

The table on the slide shows the result: The optimized version, performing nearly identical work, runs more than **twice as fast**. This improvement demonstrates how careful structuring can mitigate the performance impact of dynamic calls.

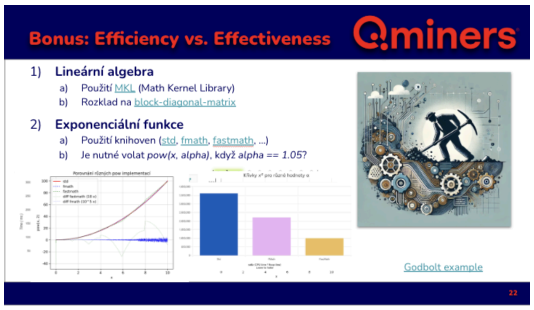
Key Takeaways

When dynamic calls are unavoidable:

1. **Avoid placing them in hot loops**. Consolidating work into fewer calls can lead to significant performance gains.
2. **Rely on measurements** to guide decisions, as modern architectures often defy intuitive expectations.

Effectiveness vs. Efficiency: Practical Examples

In performance-critical systems, efficiency often takes center stage, but it is equally important to consider **effectiveness**—whether the chosen solution is truly appropriate for the task at hand. Below are examples where an overly narrow focus on efficiency risks missing the broader context of effectiveness.



Bonus: Efficiency vs. Effectiveness

Qminers

1) **Lineární algebra**
   a) Použití MKL (Math Kernel Library)
   b) Rozklad na block-diagonal-matrix

2) **Exponenciální funkce**
   a) Použití knihoven (std, fmath, fastmath, …)
   b) Je nutné volat *pow(x, alpha)*, když *alpha == 1.05*?

Godbolt example

22

Porovnání různých pow implementací
std
fmath
fastmath
diff fastmath (10 x)
diff fmath (10^5 x)
Křivky x^α pro různé hodnoty α

1. Linear Algebra: Matrix Multiplication

- **Efficiency**: Libraries like Intel's Math Kernel Library (MKL) are optimized for modern hardware and excel at processing large matrices, making them highly efficient in such scenarios.
- **Effectiveness**: For small matrices or vectors (e.g., fewer than 16 elements), the constant overhead of calling MKL (e.g., dynamic dispatch and initialization) can outweigh the benefits of its sophisticated algorithms. In such cases, a simple, naive implementation is faster and better aligned with the task. Is MKL truly the right tool for small matrices?
- **Further Context**: If the matrix contains a large number of zeros (e.g., block-diagonal structure), even a highly optimized full matrix multiplication becomes wasteful. Effectiveness calls for recognizing such patterns and replacing the operation with smaller, targeted multiplications of individual blocks, which directly address the problem without unnecessary work.

2. Exponential Functions (Godbolt example: https://godbolt.org/z/qKPaaaxhx)

- **Efficiency**: Standard implementations like `std::exp` provide exceptional precision, but this comes at the cost of performance. Approximate alternatives such as `fmath` or `fastmath` offer significant speedups, trading off a small degree of accuracy for practical gains in efficiency.
- **Effectiveness**: Even approximate functions may be unnecessary in some cases. For example, if an analyst specifies an exponential function with an adjustable exponent (α\alphaα), but calibration reveals α ≈ 1.05, does the problem truly require an exponential? A simple linear function could serve just as well within the relevant range, avoiding the computational overhead entirely. This reflects the essence of effectiveness: asking whether the solution truly fits the problem.

Key Insights

1. **Effectiveness First**: Before pursuing efficiency, confirm that the chosen tool or approach is the right one for the task. Overlooking effectiveness can lead to highly optimized solutions to problems that don't need solving in the first place.
   - Is the constant overhead of a tool like MKL justified for small inputs?
   - Is a highly optimized full matrix multiplication sensible for sparse or structured matrices?
   - Is an exponential function truly required, or could a simpler alternative suffice?
2. **Efficiency Within Context**: Once the approach is validated as effective, efficiency can be optimized by leveraging appropriate tools and techniques.

By prioritizing effectiveness, performance-critical systems can avoid wasteful over-engineering and focus on solutions that are both appropriate and efficient.