High-Frequency Trading and Ultra Low Latency Development Techniques

Nimrod Sapir
VP R&D
qSpark Ltd
nimrod.s@qspark.co
https://www.linkedin.com/in/nimrodsapir/

qSpark

About me

- Been working in the HFT world for the last 5 years
- Worked on performance sensitive code for most of my career (mostly for storage systems)

What am I going to talk about ?

- What is algo trading and high-frequency trading
- How does high-frequency trading impacts the market
- Main challenges in designing high-frequency trading infrastructure
- Development techniques used in qSpark for the world of high-frequency trading

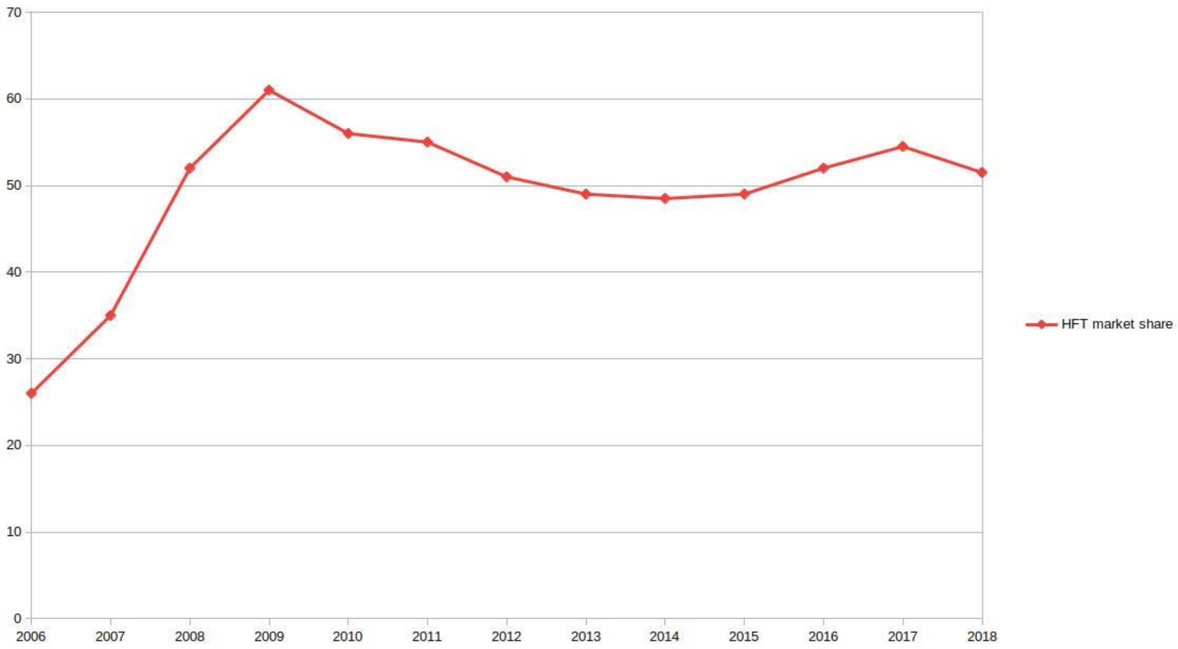
What is algorithmic trading/High Frequency trading

- Algorithmic trading is any software which follow a predefined algorithm to place trading instructions
- High-frequency trading is algorithmic trading characterized with very high trading rate and short investment horizon.
- Usually, HFT algos do not try to predict overall long term market behaviour (i.e. will it go up or down)
- HFT algorithm profitability is dependent on its ability to perform trading actions at critical points in time in an extremely latency-sensitive manner.

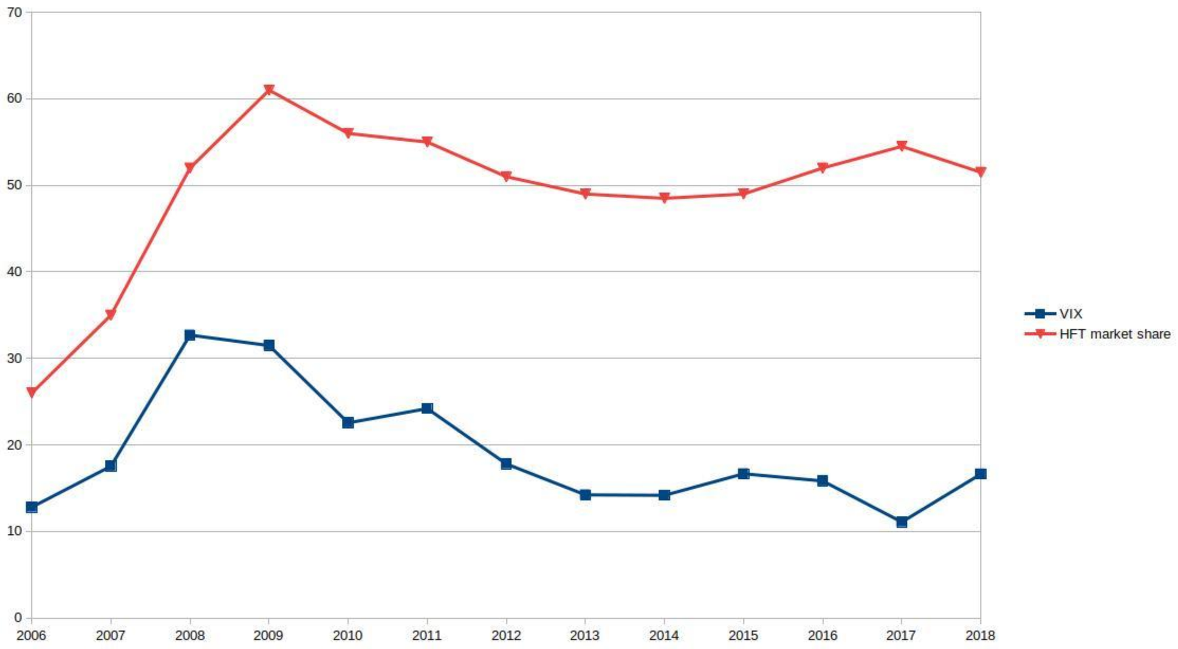
High-frequency trading market share (estimation)

- Although there is no official statistics, HFT is estimated to account to at least 50% of the US equity (shares) trading volume
- Notice that trading volume does not equal capital
- The market share of HFT has declined, as did profitability, since the peak year (2009)

High frequency trading market share (estimation)

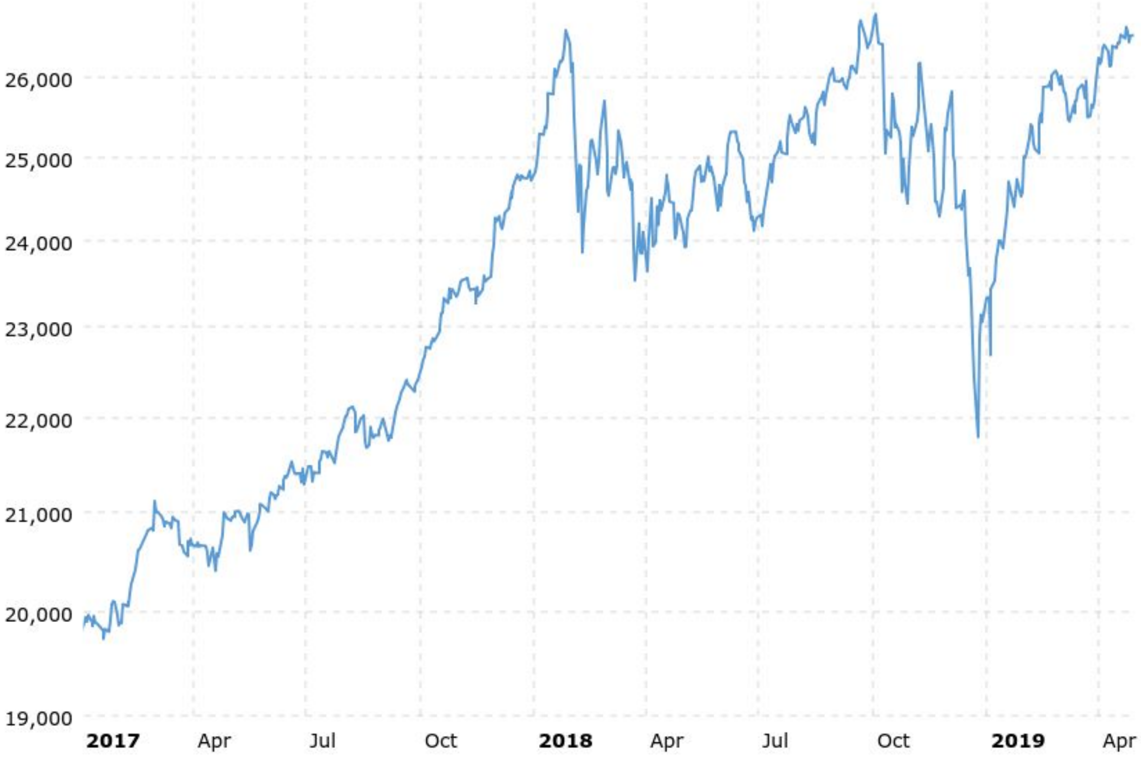


High frequency trading market share (estimation)



VIX = Volatility index. Reflects the level of anxiety in the market.

High frequency trading market share



Dow Jones - 2017-2018

Is HFT good for the economy

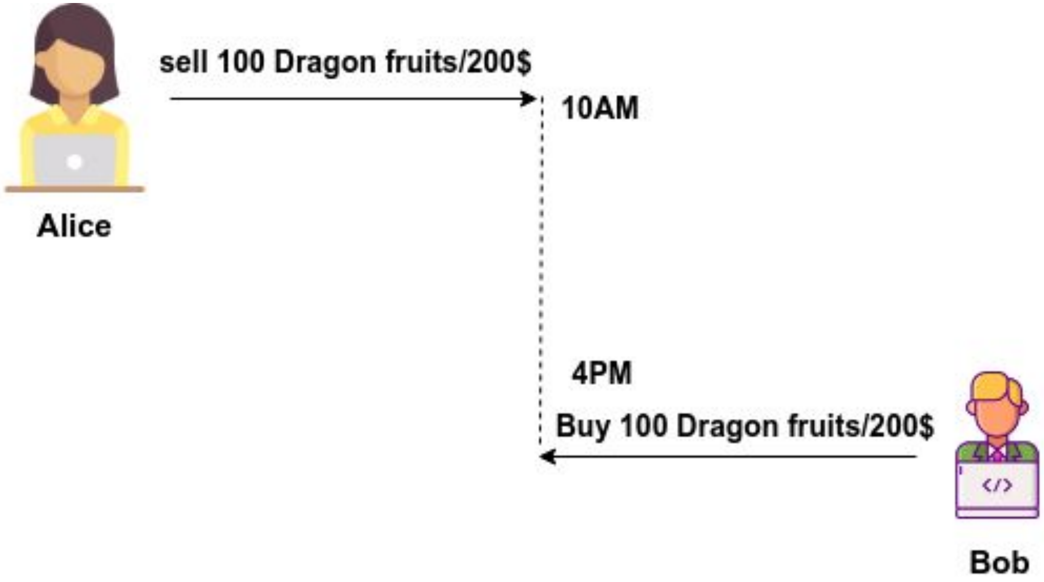
- The questions on high-frequency trading impact on the economy is old as HFT itself
- Many concerns rose around events such as the “flash crash” of 2010
- Two known benefits of HFT are improvement of market liquidity and narrowing of bid-offer spread

Market making

- Alice comes to the market and would like to buy apples
- Apples are widely available in the market, and there are always buyers and sellers for them
- However, in order to get funds, she need to sell the dragon fruits she already has - which are a not very popular!

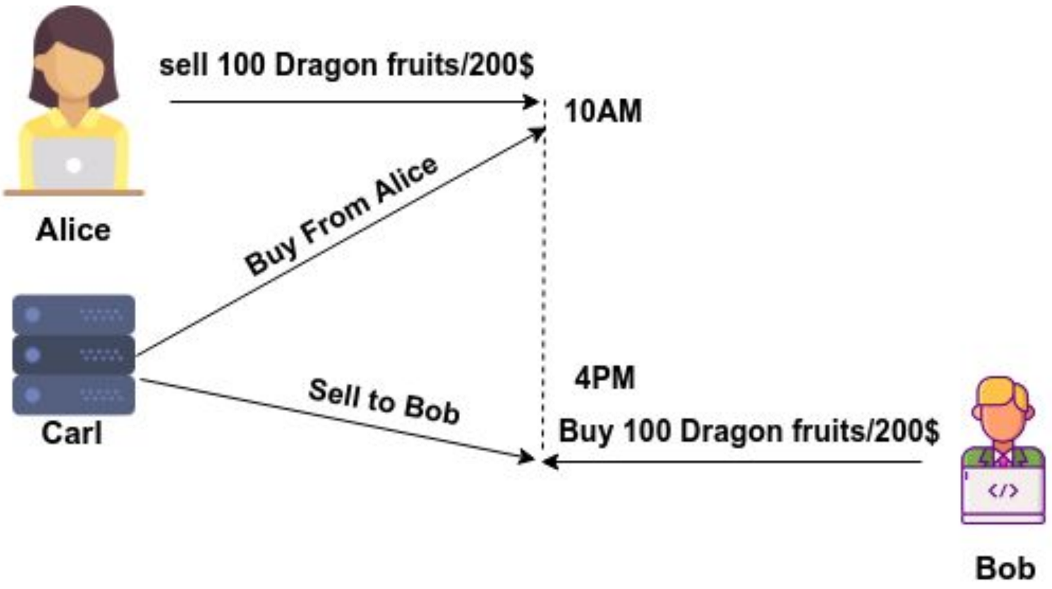
Market making

- For actual trade to happen, alice has to hold her order open for 6 hours
- During this time she may decide to give up (for example, if price of apples rises and she can't use those funds anymore)



Market making

- Carl, the trading algo is willing to match Alice sell, allowing her to buy apples immediately
- Carl will wait until Bob shows up and complete the transaction



Market making

- Carl is considered a “market maker” as it enabled all the transactions, which may not have happened without it.
- Although it seems that Carl did not earn from the transaction, in the stock market it will actually earn rebates, which are percentages of the fees the market took from each transaction.
- However, Carl also took a large risk, as there is no guarantee that a buyer would come in, or that the price wouldn’t drop

Wait, but what is fast?

- The definition of HFT is very vague, and is constantly changing
- At the beginning of the 21st century - a turnaround of seconds would be considered “high frequency”
- Today, measurements is in microseconds

Wait, but what is fast?

- Light travels at **300 m/microsecond**
- Forget about running your HFT rig from your own data center or from the cloud
- Also, the days of looking for the closest location to the exchange are over - you must reside **inside** the exchange

Market fairness and network latency

- Due to strict market regulation, any difference in external network latency was evened out.
- Therefore, the real competition now resides inside the traders' technological stack

NYSE

HFT trading infra - main challenges

...And they will lead you through the dark ot the widest, deepest river of wealth ever known to man. You'll be shown your place on the riverbank, and handed a bucket all your own. Slurp as much as you want, but try to keep the racket of your slurping down....

- Kurt Vonnegut -

HFT trading infra - main challenges

- A good trading product has the right balance of profitable trading logic, strong trading infrastructure and ability to quickly act and react to market events
- Nothing will save you if your trading logic is misguided or if you are slow to react - you live and die by your technical stack.
- Brand is non-existing - nothing prevents a smarter or quicker competitor from taking over your “market share”
- All actions must comply to strict market regulation - bugs can easily result in fines!

HFT trading infra - main challenges

- When talking about the competition in the HFT world, you may imagine something like this:

Here a better reference may be something like that:

HFT trading infra - main challenges

- The market behavior can never be accurately predicted, and you can never be certain you are going in the right direction
- You need to be able to be extremely quick, not only to make a quick transaction, but also to be able to revert quickly and cut your losses when you are stuck with a bad trade
- You are never “fast enough”, every nanosecond you can cut of your real-time flow will result in increase in profitability, and vice-versa

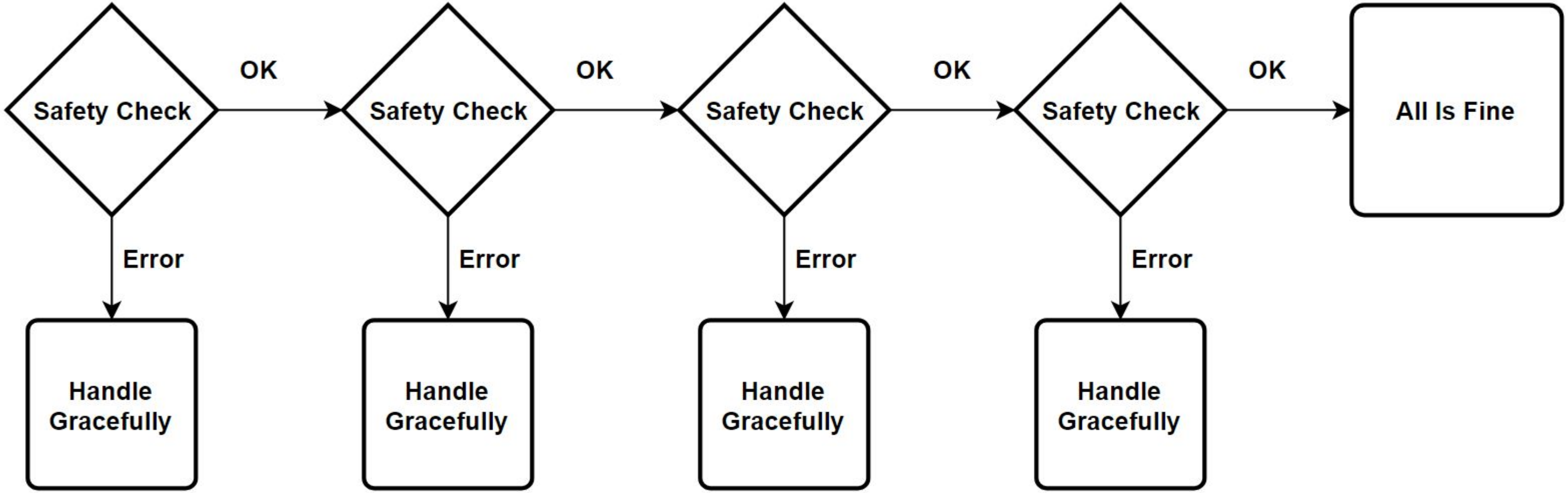
Main development approach in the qSpark trading infra

- End-to-end kernel bypass - system calls are too slow to use in real time
- Avoid context switching, queuing and data transfer between threads as much as possible
- Deterministic, static code flow, which makes as many decisions as possible in compilation time
- Minimize cache misses and wrong branch prediction
- Use custom-tailored data structures for specific use cases
- It is not faster if you haven’t measured it

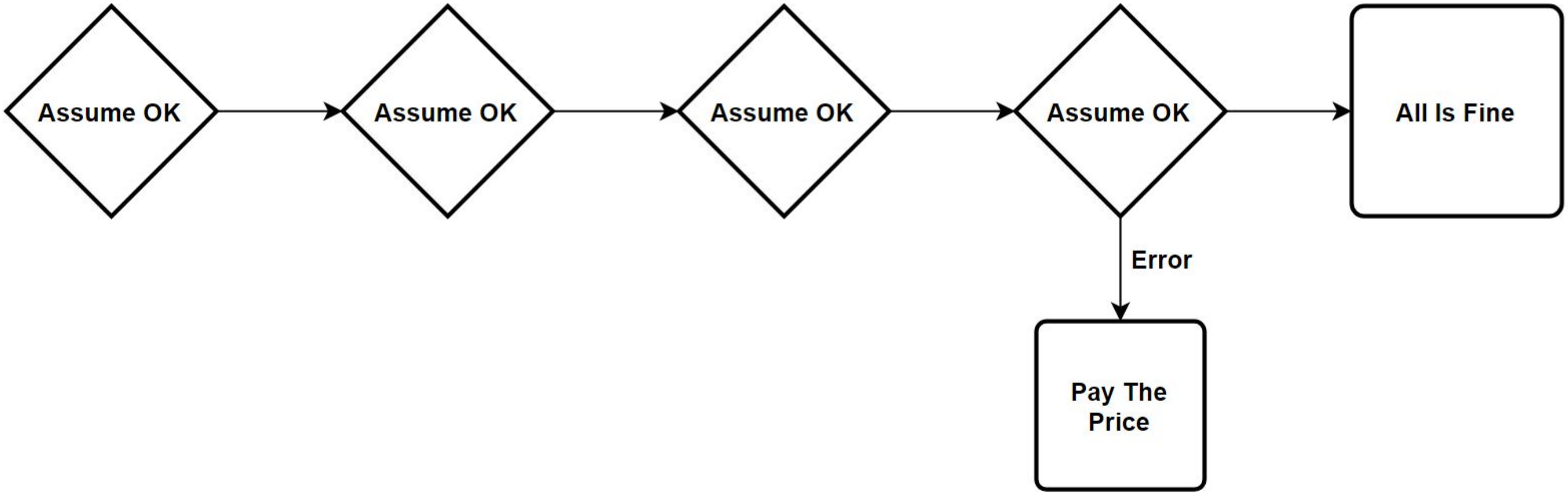
Deterministic code flow and branching minimization

- Every run-time action has a performance penalty
- This is worsen by the fact in the case of branch misprediction which means wasted CPU cycles
- Our design strives to create a deterministic, static flow, which minimizes runtime branching by moving overhead to compilation and initialization time

Deterministic code flow and branching minimization



Deterministic code flow and branching minimization



Compile time polymorphism using CRTP (Curiously recurring template pattern)

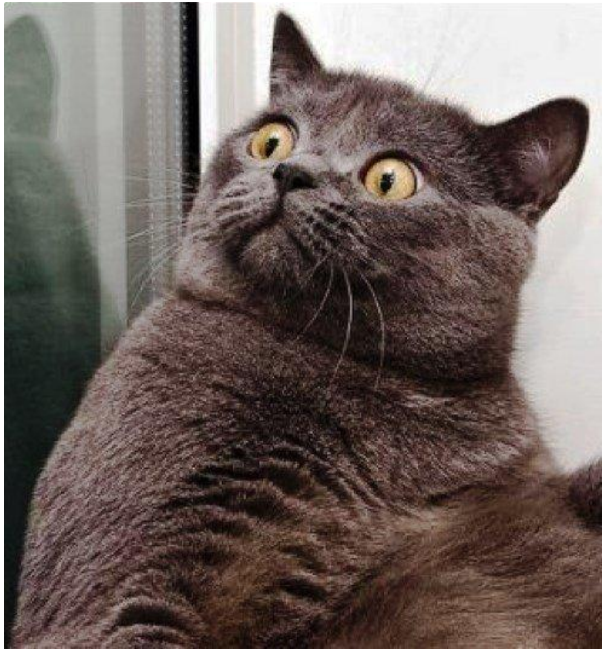
```
class order
{
   virtual void place_order() {// Generic implementation...}
};
class specific_order : public order
{
   virtual void place_order() override
    {// Specific implementation...}
};
class generic_order : public order {// No implementation};
```

Compile time polymorphism using CRTP (Curiously recurring template pattern)

```
template <typename actual_type>
class order
{
void place_order() {static_cast<actual_type*>(this)->actual_place();}
void actual_place() { // Generic implementation… }
};
class specific_order : public order<specific_order>
{
   void actual_place() { // Specific implementation... }
};
class generic_order : public order<generic_order> {...};
```

Compile time polymorphism using CRTP (Curiously recurring template pattern)

```
template <class Execution, template <class A,
class B> class SocketHandlerType =
SocketHandlers::Tcp>
class BOE2 : public ExecutionProtocol<Execution,
SocketHandlerType, BOE2SequenceSourceType,
HeartbeatPolicy::Send>
{ ... }
```



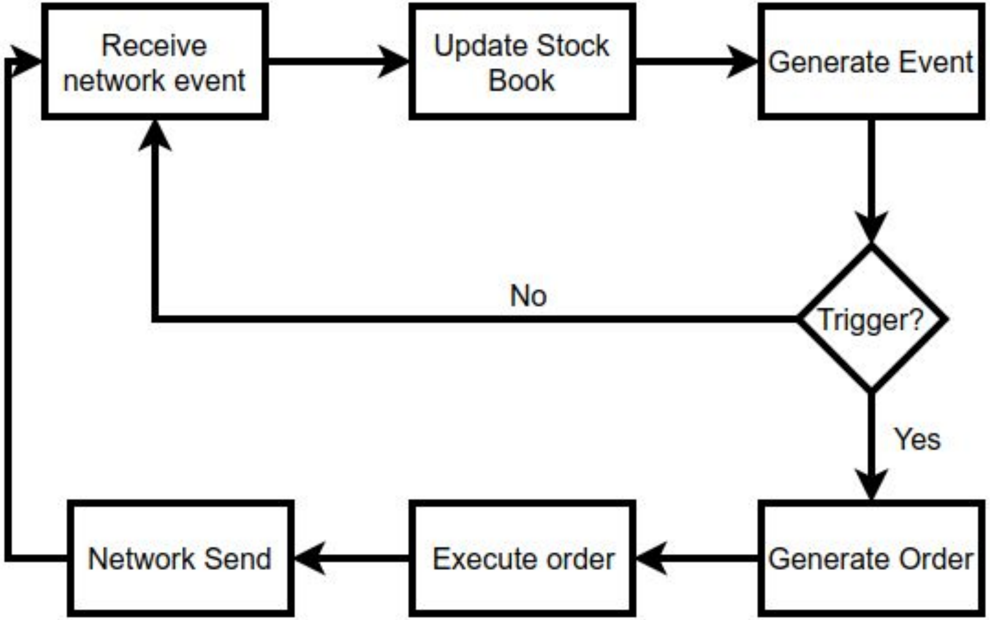
Deterministic code flow and branching minimization

- There are many more techniques that can be used for achieving a more deterministic code flow
  - Maps with static values which can be evaluated in compile-time
  - Compile-time configuration which replaces runtime flags with templated values
  - Rearranging and/or statements order to move the more predictable values first
  - Of course: constexpr all the things!
- Those techniques may have their own cost - mostly in compile-time, but sometimes also in run-time (code bloat)

Warming up the cache

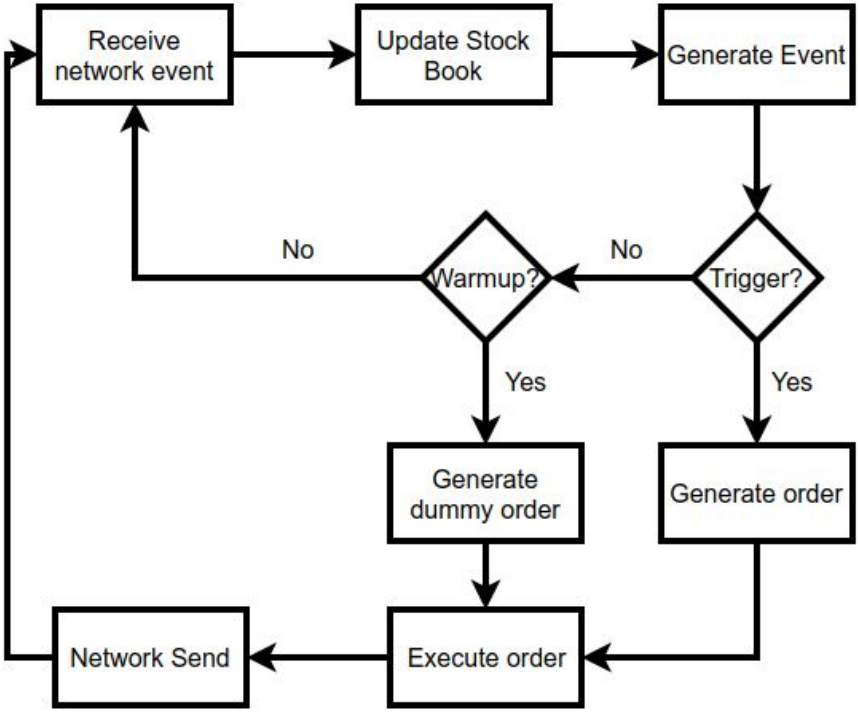
- Cache misses are one of the highest overhead for a low-latency code.
- However, cache is very unpredictable - in multi-threaded environment, there is a constant fight for the L3 cache
- This is worsen by the fact that the most critical flow is sometimes extremely rare

Warming up the cache



- When trigger actually occurs, the likelihood of the order placement flow to be in the cache is extremely low
- In addition, branch prediction will assume order is never sent

Warming up the cache



- Now the order placement flow is way likelier to be in the cache, and branch prediction is more balanced
- Sounds simple, but there are many complications

Warming up the cache

```
size_t g_total_value{};

void add_order_value(Order& order)
{
    g_total_value += order.get_amount() * order.get_price();
}
```

- There is a side-effect here, that we need to eliminate
- Naive approach, let's check if the order is warming only

Warming up the cache

```
size_t g_total_value{};

void add_order_value(Order& order)
{
    if (!order.is_warming)
        g_total_value += order.get_amount() * order.get_price();
}
```

- We may have made things way worse!
- Multiple mispredictions can easily lead to warming actually adding performance penalty

Warming up the cache

Warming up the cache

```
std::array<size_t, 2> g_total_value{};

void add_order_value(Order& order)
{
    g_total_value[order.is_warming] += order.get_amount() * order.get_price();
}

size_t get_order_value(){ return  g_total_value[false] };
```

- The misprediction is eliminated
- Although we are “warming” the wrong entry in the array, locality makes it very likely that we are actually warming both

Warming up the cache

- This is a very simple example, but actually avoiding all side-effects in a complicated flow, without skipping any part of the code, may be very challenging and may require major redesign
- Any bug here would (and did) lead to serious issues
- Therefore, handle with care!
- That said, in the world of micro optimization, cache warming is extremely efficient!

Tailor-made data structures for specific use cases

- Our code contains many data structures which are optimized for specific use cases
- Some are extremely general and complex, some were made specifically to resolve specific issues
- Here is one simple example: static_flat_map

Static Flat Map

```
void foo(std::map<...>& multi_threaded_small_map,
            lock_type& very_busy_lock)
{
    std::lock_guard<...> guard(very_busy_lock);
    for (auto& item : multi_threaded_small_map)
    {
        ...
    }
}
```

Static Flat Map

```
void foo(std::map<...>& multi_threaded_small_map,
             LockType& very_busy_lock)
{
    std::map<...> local_map;
    {
        std::lock_guard<...> guard(very_busy_lock);
        local_map = multi_threaded_small_map;
    }
    for (auto& item : local_map) {...}
}
```

Static Flat Map

- Keep a sorted array
- Use binary search to find items
- Result: Much better performance for copying and iterating over a small map with known size

Static Flat Map

```
void foo(static_flat_map<...>& multi_threaded_small_map,
            lock_type& very_busy_lock)
{
    static_flat_map<...> local_map;
    {
      std::lock_guard<...> guard(very_busy_lock);
      local_map = multi_threaded_small_map;
    }
    for (auto& item : local_map) {...}
}
```

Static Flat Map

	std::map	StaticFlatMap
Copy Time	**~25usec**	**1usec>**

Static Flat Map

Pros:

- Quick iteration
- Quick copy
- Quick lookup and edit
- Sequential and Static

☺

Cons:

- Slow insert and remove
- Limited - size must be known in advance
- Not as good for large maps
- Not as good for large objects

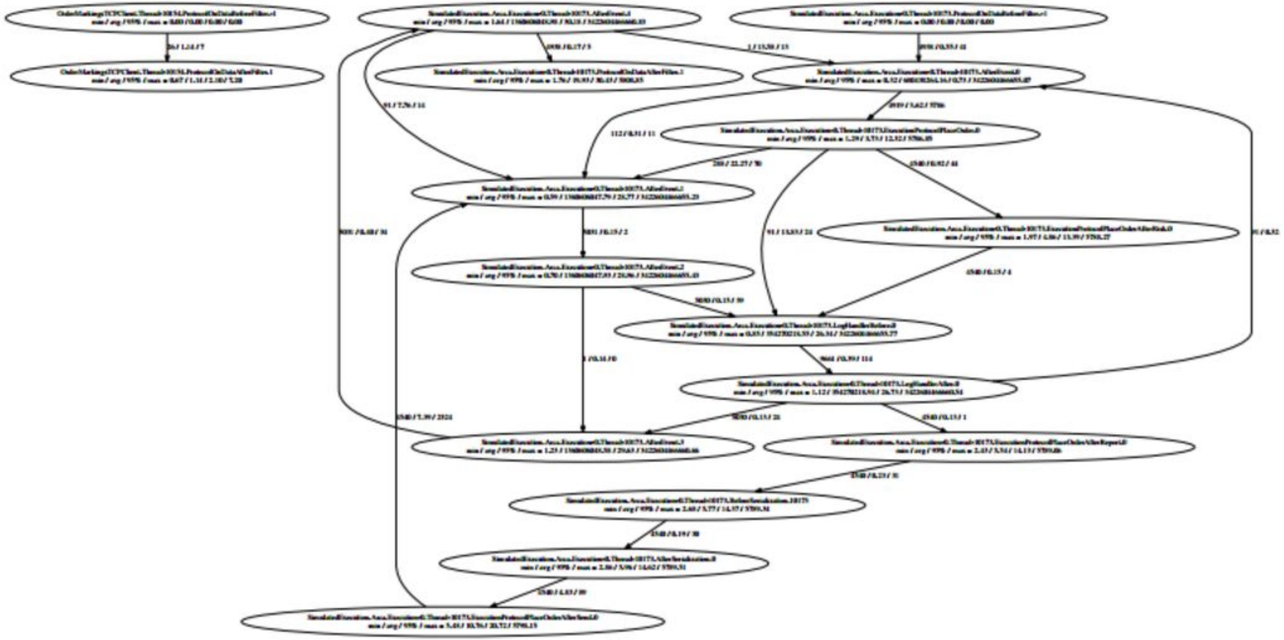
☹

https://github.com/DanielDubi/StaticFlatMap

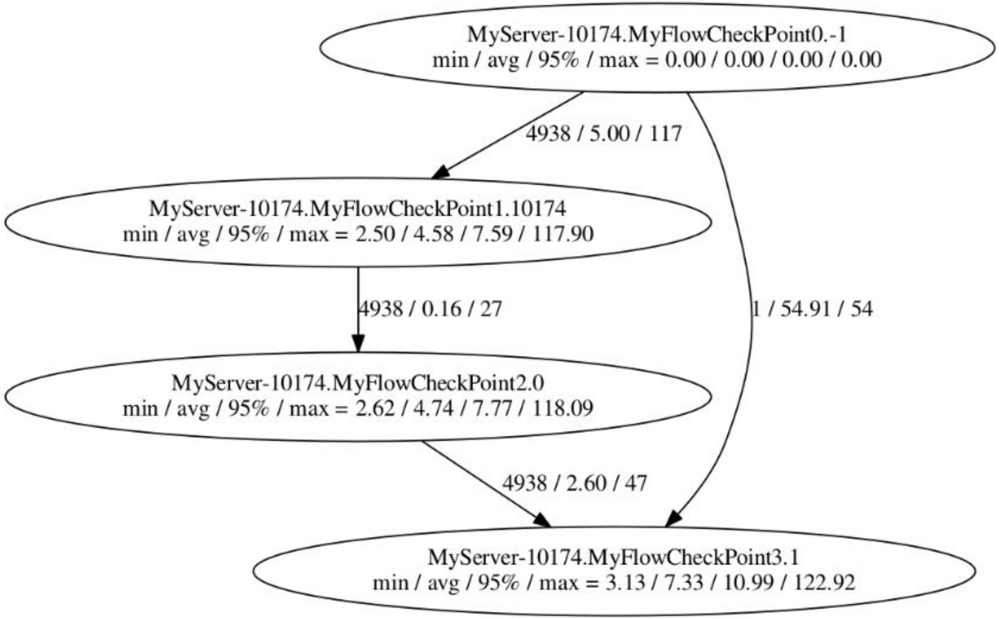
Performance measurement

- When it comes to micro optimization and ultra low-latency, no guarantee for “better average performance” is acceptable as-is.
- This means that each performance tweak has to be continuously measured to show better performance.
  - For example, -march and -mtune flags actually degraded performance in our environment.
- However, testing the entire matrix of possible combinations is practically impossible.

Performance measurement



Performance measurement



Performance measurement

- This performance measurement technique is nice, but it has a lot of overhead
- The minimal overhead is simply taking a timestamp. For example, in our environment:

Timestamp accuracy	150 nanosecond	1 microsecond
Timestamp taking overhead	50 nanoseconds	10 nanoseconds

- We have to separate lightweight real time counters from intrusive measurements used in production

Disclaimer

- Premature optimization is the root of all evil (Donald Knuth)
- Premature micro-optimization is just plain stupid!
- The techniques described all have very serious costs, pitfalls and trade offs
- Use with care, and only when micro-optimization is required

We are hiring C++ developers!

nimrod.s@qspark.co

https://www.linkedin.com/in/nimrodsapir/

THANKS!

Any questions?

Static Flat Map:

https://github.com/DanielDubi/StaticFlatMap